Implementation of Character Controls and Combat System in the Action Adventure 'Scout COD' realized with Unity 3D

BACHELOR THESIS

for obtaining the degree 'Bachelor of Science'

Author:	Sarah Stumfol
Matriculation number:	171111
Degree course:	Medien und Informationswesen
University:	Hochschule Offenburg
Supervisors:	Prof. Sabine Hirtes
	Prof. Dr. Volker Sänger
Date:	April 8, 2014

Statutory Declaration

I declare that I have written this thesis independently, that I have not used any other than the stated sources/ resources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

---

place, date, signature

Acknowledgements

My bachelor thesis has benefited greatly from the support of numerous people, some of whom I would sincerely like to thank here.

First of all, I want to say thanks to my project team which consists of Max Schumayer, Dominik Kirner, Jennifer Lehmann, Melanie Schweis, Benjamin Kreft and Marco Rappenecker. Without them, this project would has not been possible to implement.

I'm also very grateful both my supervisor Prof. Sabine Hirtes and my advisor Prof. Dr. Volker Sänger for their guidance during the creation of this thesis.

Sincere thanks are given to Andreas Nolle and Robert Kuti for proofreading.

Finally, I want to express my gratitude to my parents Peter and Iris Stumfol. They supported me greatly since I have started my studies. My special thanks goes to my father who was my personal advisor regarding technical and computer science issues.

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Motivation

Video games shape our culture and influence all kind of people of our modern societies. Nowadays, they are widely spread and an important form of entertainment. The fundamental elements of a game are the look and feel and also the interactivity which excite people of all ages. In many nations an individual industry developed and its sales even overtake those of the film industry. Video games belong to the most influential leisure activities of the 21st century.

Commercial game development began in the 1970s with the introduction of first generation game consoles and home computers. Due to low costs and low capabilities of the hardware, a lone programmer could develop a full game. However, these days, ever-increasing computing power and heightened consumer expectations made it difficult for a single developer to produce a mainstream game. Quite the contrary, a major team works nowadays on the design and development of a game.

In tandem with the growth of the size of development teams in the industry, the development of costs have increased. Development studios need to be able to pay their staff a competitive wage in order to attract and retain the best talent, while publishers are trying constantly to keep costs to a minimum in order to turn a profit on their investment. Typically, a game development team can range anywhere from five to 50 people, sometimes exceeding 100. And even this number is not the last margin.

Most modern console or PC games take from one to three years till completion, where as a mobile game can be developed in a few months. This duration is influenced by a number of factors, such as genre, scale, development platform and amount of assets.

Game development is a process that starts with the concept and ends at the release of the game. Game programming represents the software development of video games. It requires substantial skills in software engineering as well as specialization in one or more of the following areas, which can also overlap heavily: simulation, computer graphics, artificial intelligence, databases, stage design, physics, audio programming and input.

All in all, games are a fascinating art form to combine video, audio, art and storytelling. The gaming industry grows and grows. It will stay pulsating and innovative games will surprise the audience again and again.

1.2 Job Definition

The present thesis deals with the implementation of the prototype for the 3D Action Adventure *Scout COD*, which includes character controls and the combat system. In short, it plays in a devastated world, in which the player personalizes a Scout who needs to survive and save other survivors. The reason for the desolation is the creation of the so-called 'managarms' which are a big and dangerous type of insects. On his journey, the players goal is to find out the reason for the formation of those beasts.

The prototype was developed in cooperation with a six-man team of Hochschule Offenburg whose tasks were concept art, character modelling, character animation, level design, sound design and programming. The present project includes the movement and combat system. The whole team will connect all components to a functional entity in the future.

The game development of *Scout COD* was realized with the well-known game engine Unity 3D. C♯ serves as the main scripting language. The target platform is Windows and the input devices are mouse and keyboard.

1.3 Short Description

In order to develop a game, an engine is required. What game engines actually are, is explained in Chapter 2. These engines comprehend a huge architecture system. Each of its components is circumstantiated. Finally, five popular game engines are compared and evaluated by means of defined criteria.

Chapter 3 explains the terms of architecture and design patterns in software development. Afterwards, those patterns are applied to the game development.

The game engine Unity 3D provides a lot of features. First of all, Unity's interface and terms are described. Then, those features which are utilized in the implementation of 'Scout COD' are exemplified in Chapter 4.

Chapter 5 represents the game 'Scout COD'. The requirement definitions for the implementation are the topic of this chapter.

Before the programming works itself, patterns are used in order to give an overview of the architecture of the complex system. All classes are presented in Chapter 6.

Chapter 7 finally deals with the proper implementation of *Scout COD*. With the help of scripting code cutouts, the most important functionalities of the implementation are demonstrated in this document.

Part II

Foundations

Chapter 2

Game Development

First of all, the question that arises is what a game actually is and which basic elements makes a game to a game.

The development of a game is divided into five steps. Each of them requires different employees that work together as a unit.

Then, the question arises what a game engine is. Besides the answer to that question given in terms of a definition, the next section also provides a historical brief overview of the history of such engines which just started about 20 years ago.

A game engine has a complex runtime engine architecture with many individual components, which are in turn divided into subsystems. Especially the next part explains the most important components, such as rendering, physics and animation.

Which kind of data is developed during game development is described afterwards.

Every game genre is structured differently and requires special technical services. Therefore, game engines differ depending on the genre for which they are designed for. They have different priorities for technologies like various animation, rendering, physics and camera systems. The technical requirements for an Action Adventure are explained in this section.

There are many scripting languages that are preferably used in game development. The most famous languages in the game industry, will be discussed in the subsequent section.

The question that I finally like to address is which game engine is suited best for someone who wants to develop a game. With several identified criteria five popular game engines are compared and evaluated. A short summary of the engine Unity 3D closes this chapter.

2.1 What is a Game?

2.1.1 Definition

People think they know what a game is because they can feel it when they start to play it. But if someone wants to develop a game, he or she realizes that this is not as easily done as imagined. The resulting game may be feels uncompleted or does not make any fun. Therefore, the challenging question arises, what a game actually is.

Many people argue that a game is the opposite of seriousness. Games should make fun. But a lot of gamers take a game very seriously indeed and get angry when they loose. One of the most important game theorizer is Johan Huizinga[1] who defines a game as 'holy seriousness'. This term characterizes the immersion and oblivion of the reality which distinguishes a game compared to a successfully game.

In the literature, there exist some formal definitions of a game. One of them is the following:

> *'A game is a system in which players engage in an artificial conflict, defined by rules, that results in a quantifiable outcome.'*[2]

Every game needs a story with an artificial conflict to pique the gamers interest and to get them tied into the virtual world. Rules define every game, they represent the boundaries and the heart of the game. They only refer to the game itself and never exist outside of it. Although a game has rules which are similar to laws, playing a game is voluntary and cannot be forced on the players. Whoever plays a game, voluntarily binds himself to the rules. All actions of the player that are defined by those rules cause a definitive result in terms of miscellaneous goals.

2.1.2 Basic Elements

The numerous elements which compose a game can be classified into four basic elements[3]:

1. *Mechanics* describe cycles and rules in a game. They define goals and how the player can or can not achieve them and what happen when they finally got it.

2. The *Story* determines the sequence of events in a game. This sequence can be linear and permanent provided or events branch and develop caused by specified actions.

3. *Aesthetics* defines the 'look and feel' of a game which is an enormously important aspect of game design because it has directly influence on gaming experience.

4. *Technology* comprises all instruments and interaction measures that are needed for developing a game. In essence, it is the medium that tells the story, transports aesthetics and accentuates all mechanics.

---

[1] HUIZINGA (2001).
[2] SALEN/ZIMMERMAN (2003).
[3] cp. SCHELL (2012), pp. 79-85.

2.2 Game Development Workflow

Game Development passes through five different phases[4] (see Figure 2.1).

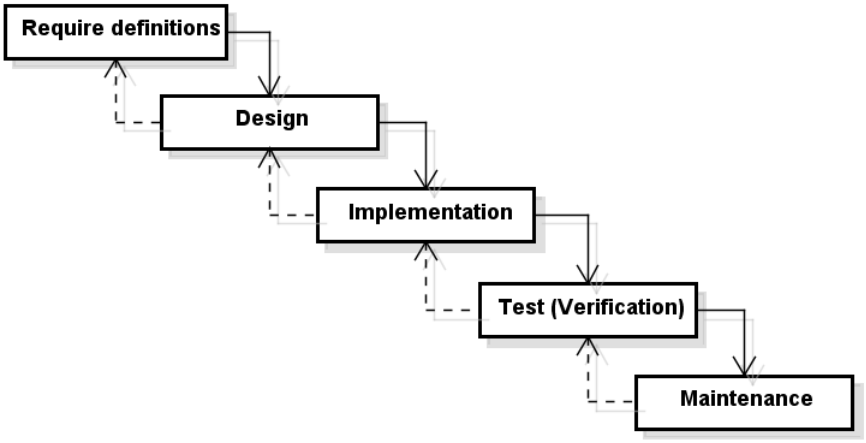


Figure 2.1: Waterfall model

1. *Requirement definitions*: An extensive document, called *performance specification*, describes all features and modules of a game in terms of requirement definitions. *Game designers* design the interactive portion of the player's experience (*gameplay).* This task includes, for example, the determination of the story, the overall sequence of chapters, levels and high-level quests.

2. *Design*: The story, quests, characters, the look and all other components relating to the game are finally elaborated in this step. *Concept artists* produce sketches and paintings that provide the team with a vision of what the game will look like. They begin their work early in the concept phase of development. At the same time, *programmers* concern themselves with the design of the implementation (more details in Chapter 3) and start prototyping.

3. *Implementation*: With the help of concept art for characters and elements of the virtual world and mood paintings, all game data is created by *artists* and *designers.* This includes 3D modeling, texturing and animation (for characters and environment), graphics for graphical user interfaces (GUI), sound design and texts (menu, dialogue system, tips, etc.). Afterwards, all files are imported into a game engine where they interact together via scripts. Character controls, quests and all other game play features are implemented by *programmers.*

4. *Test (Verification)*: A plenty of different methods for testing the features help to bug-fix, optimize and verify the game. *Professional testers* and *beta testers* are responsible for this job.

5. *Maintenance*: If these steps are finally passed through, the game can be released. Henceforward, the game needs to be maintained continuously. Especially game technology improves very fast so that new features can upgrade the performance.

[4]cp. REHFELD (2014), pp. 38-59.

2.3 What is a Game Engine?

2.3.1 Definition

In the literature exist several definitions for the term 'game engine'. The following quotation puts the content of such definitions in a nutshell.

> *'The term game engine is used to describe a set(s) of code to build a gaming application. A game engine is more specially a framework comprised of a collection of different tools, utilities, and interfaces that hide the low-level details of the various tasks that make up a video game*[5]*.'*

Thus, a game engine is a special framework for video games that controls the game process and is responsible for the visualization. Developers use this framework for create games for video game consoles, mobile devices and personal computers. The core functionality typically provided by a game engine includes a lot of components that are defined in Chapter 2.4. The process of game development is often economized by reusing and adapting the same game engine to create different games or to make it easier to 'port' games to multiple platforms.

2.3.2 History

The term 'game engine' arose in the mid-1990s with reference to first-person shooter games like the very popular game called *Doom* by id Software. *Doom* was designed with a reasonably well-defined separation between its core software components (like the three-dimensional graphics rendering system, the collision detection system or the audio system) and the art assets, game worlds and rules of play that comprise the player's gaming experience.

Subsequent games, such as id Software's *Quake III Arena* and Epic Games's *Unreal* were designed with a separately developed engine and content. There was a strong rivalty between those two companies because Epic's Unreal Engine became far more popular than id Tech 4.

Modern game engines are some of the most complex applications, often featuring dozens of finely tuned interacting systems to ensure a precisely controlled user experience. The continued evolution of game engines has created a strong separation between rendering, scripting, artwork, and level design. Threading has gained in importance due to modern multi-core systems and increased demands for realistic games. Typical threads involve rendering, streaming, audio, and physics.

2.4 Runtime Engine Architecture

Figure 2.2 shows all of the major runtime components[6] that constitute a typical 3D game engine. Game engines are built in *layers*. Usually upper layers depend on lower layers. If a lower layer depends on an higher layer, this is called *circular dependency*, but such kind of relationships should be avoided in any software system.

[5] ALLEN SHERROD (2009).
[6] cp. GREGORY (2009), pp. 28-49.
[7] Ibidem

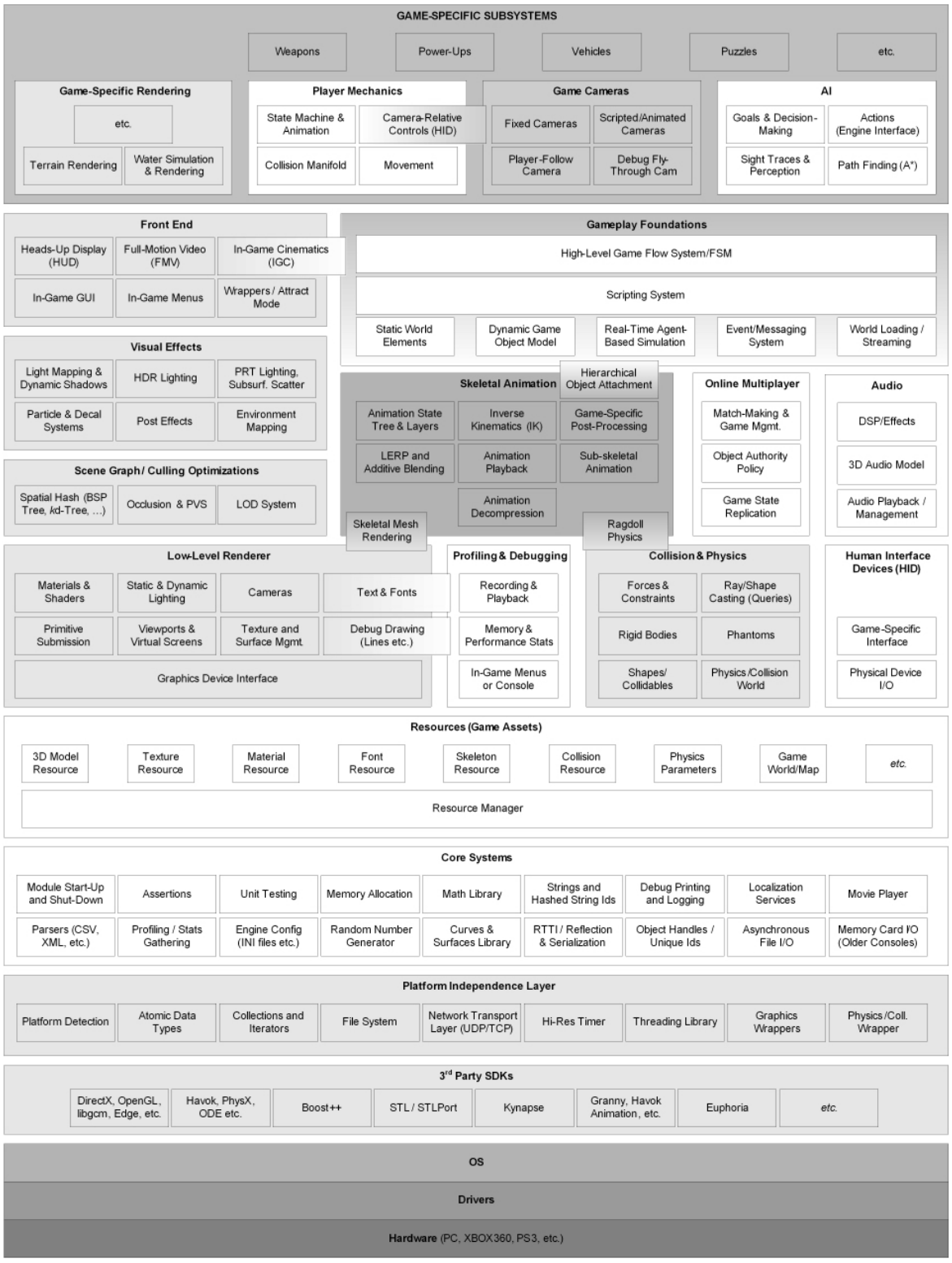


Figure 2.2: Runtime game engine architecture[7]

2.4.1 Target Hardware

Target hardware represents the computer system or console on which the game will be run (see Figure 2.3). Typical platforms are Microsoft Windows- and Linux-based PCs, the Apple iPhone and Macintosh, Microsoft's Xbox 360, Sony's PlayStation, PlayStation 2, PlayStation Portable (PSP), and PlayStation 3, Nintendo's DS, Game-Cube and Wii.

Figure 2.3: Hardware layer

2.4.2 Device Drivers

Device drivers are low-level software components provided by the operating system or hardware vendor. Drivers manage hardware resources and abstract the operating system and upper engine layers from the details of communicating with the myriad variants of available hardware devices (see Figure 2.4).

Figure 2.4: Device driver layer

2.4.3 Operating System

The *operating system (OS)* orchestrates the execution of multiple programs on a single computer. The OS layer is shown in Figure 2.5. Operating systems like Microsoft Windows implement a time-sliced approach to share the hardware with multiple running programs, known as pre-emptive multitasking. This means that a PC game or even a software can never assume to have full hardware control.

Figure 2.5: Operating system layer

2.4.4 Third-Party SDKs and Middleware

Most game engines contain some *third-party software development kits (SDKs)* and *middleware*[8](see Figure 2.6). The functional or class-based interface provided by a SDK is called application programming interface (API).

Some examples for such third-party components are data structures and algorithms, graphics, collisions and physics, character animation, artificial intelligence (AI) and biomechanical character models.

[8]In game development, middleware defines a subsystem for partitions like game physics.

Figure 2.6: Third-party SDK layer

2.4.5 Platform Independence Layer

The majority of game engines are required to be run executable on more that one hardware platform. Most companies always target their games at the wide variety of platforms, because opens up the largest possible market. Those companies that do not target at least two different platforms per game are first-party studios[9]. Therefore, most game engines are architected with a *platform independence layer* (see Figure 2.7). This layer is located on the top of hardware, drivers, operating system and other third-party software and abstracts the rest of the engine from the most of the knowledge about the underlying platform.

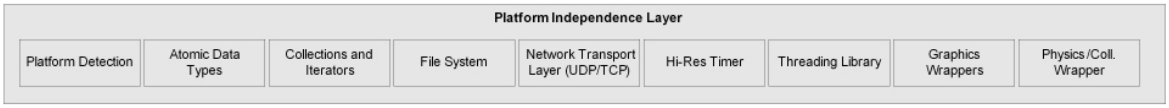


Figure 2.7: Platform independence layer

2.4.6 Core Systems

Every game engine requires a collection of useful software utilities. The task to provide such utilities is part of the *core systems*. Here are some examples of facilities the core layer provides:

- *Assertions*: Assertions are error-checking code snippets that are inserted to catch logical mistakes and violations of the programmer's original assumptions. Assertion checks are generally stripped out of final production build of the game.
- *Memory management*: Virtually every game engine implements its own custom memory allocation system to ensure high-speed allocations and deallocations and to limit the negative aspects of memory fragmentation.
- *Math library*: Every game engine has one or more math libraries due to the reason that mainly based on mathematics. They provide possibilities for vector and matrix math, trigonometry, numerical integration and many others.
- *Custom data structures and algorithms*: A suite of tools for fundamental data structures (like linked lists or binary trees) are required during a game production.

A typical core systems layer is depicted in Figure 2.8.

2.4.7 Resource Manager

The *resource manager* (see Figure 2.9) provides a unified interface for accessing all types of game assets and other engine input data. Some game engines do this in a highly central-

[9]First-party studios are studios that belong to one console producer (for example Nintendo).

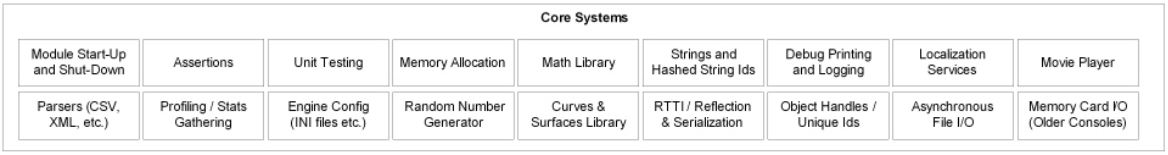


Figure 2.8: Core engine systems

ized and consistent manner, others take an ad-hoc[10]approach, often leaving it to the game programmer to directly access raw files on disc or within compressed archives.

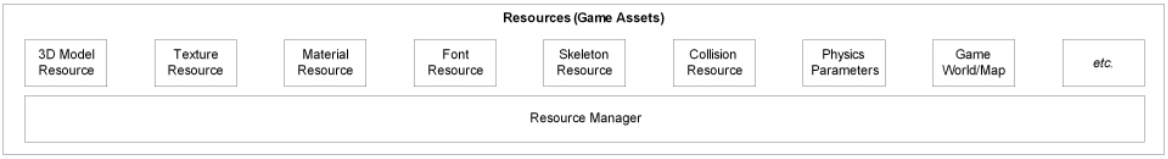


Figure 2.9: Resource manager

2.4.8 Rendering Engine

The *rendering engine* is one of the largest and most complex components of a game engine. Renderers can be designed in many different ways. Most rendering engines share some fundamental design philosophies, largely influenced by the design of 3D graphics hardware on which they depend. One common and effective approach for designing rendering engines is to utilize a layered architecture as follows.

Low-Level-Renderer

Low-level renderer encompasses all raw rendering facilities of the engine. A collection of geometric primitives have to be rendered as efficiently as possible and without taking into account for which parts of a scene may be visible. This component is separated into various subcomponents (see Figure 2.10).

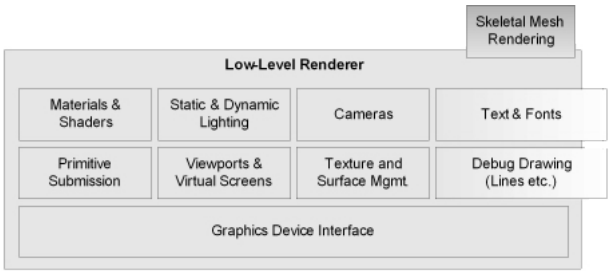


Figure 2.10: Low-level rendering engine

Graphic SDKs[11]require a reasonable amount of code that have to be written just to enumerate the available graphic devices, initialize them, set up render surfaces and so on. This is typically handled by a component for that every game engine has its own terminology. A summarized definition are *graphics device interfaces*.

[10]An ad hoc net connects two or more end devices to an intermeshed net.
[11]examples for graphic SDKs: DirectX and OpenGL

The other elements in the low-level renderer interact in order to collect submissions of *geometric primitives*, such as meshes, line lists, point lists, particles, terrain patches, text strings, and much more, and render them as efficiently as possible. The low-level renderer usually provides a viewport abstraction with an associated camera-to-world matrix and 3D projection parameters. It also manages the state of the graphic hardware and shaders via its *material system* and its *dynamic lighting system*. Each submitted element is associated with a material and is affected by n dynamic lights. The material describes the textures used for the primitive, which device state settings have to be applied, and which vertex and pixel shaders have to be used while rendering the primitive.

Scene Graph/ Culling Optimizations

A higher-level component is needed in order to limit the number of primitives that will be submitted for rendering, based on some sort of visibility determination. This layer is depicted in Figure 2.11.

A simple *frustum culling*[12](for example, removing objects that the camera cannot 'see') is everything what is required for small game worlds. For larger worlds, a more advanced *spatial subdivision* data structure might be used to improve rendering efficiency, by enabling a very quick and efficient determination of the *potentially visible set (PVS)* of objects. Spatial subdivisions can take several forms, including a *binary space partitioning (BSP) tree*, a *quadtree*, an *octtree*, a *kd-tree* or a *sphere hierarchy*. They are often called a *scene graph*, although technically the latter is a particular kind of data structure and does not subsume the former.

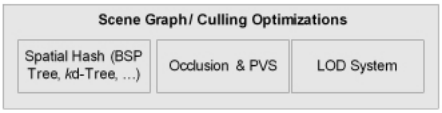


Figure 2.11: Scene graph/ spatial subdivision layer (for culling optimization)

Visual Effects

Modern game engines offer a wide range of visual effects (see Figure 2.12), such as particle systems (fire, smoke, dust, water, etc.), decal systems (for foot prints, bullet holes, etc.), light mapping and environment mapping, dynamic shadows and full-screen post effects.

Game Engines possess an *effects system* component which manages the specialized rendering needs of particles, decals and other visual effects. The particle and decal systems are usually distinct elements of the rendering engine and serve as inputs to the low-level renderer. Otherwise, light mapping, environment mapping and shadows are usually handled internally within the rendering engine. Full-Screen post effects are either implemented as an integrated part of the renderer or as a separate component that operates on the renderer's output buffer.

[12]Frustum culling is the region of space in the modelled world that may appear on the screen. It is the field of view of the notional camera.

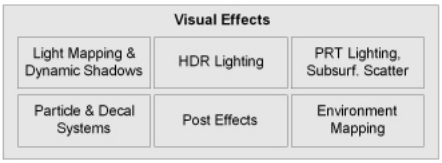


Figure 2.12: Visual effects

Front End

More often than not, a 3D game includes 2D graphics for various purposes:

- *heads-up display* (HUD) where the player can see properties like health or endurance
- in-game menus, consoles and other development tools
- in-game *graphical user interface* (GUI) allowing the player to manipulate his or her character's inventory, configure units for battle or perform other complex in-game tasks

The front end layer is presented in Figure 2.13. In general, 2D graphics like those described above are implemented by drawing textured quads (pairs of triangles) with an orthographic projection.

The *full-motion video* (FMV) system is responsible for playing full-screen movies that have already been recorded. The *in-game cinematics* (IGC) system allows cinematic sequences to be choreographed within the game itself, in full 3D. For example, during the walk of a player through a city, a conversation between two key characters might be implemented as an in-game cinematic. IGCs can include player characters. They are might done as a deliberate insection during the player has no control, or they may be subtly integrated into the game without the gamer is even realizing that an IGC is taking place.

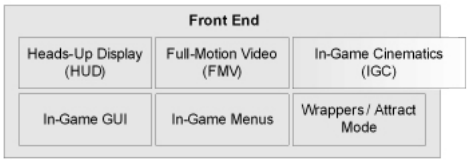


Figure 2.13: Front end graphics

2.4.9 Profiling and Debugging Tools

Games are real-time systems and game engineers often need to optimize the game performance. Memory resources are usually rare, so developers heavily use memory analysis tools as well. The profiling and debugging layer (see Figure 2.14) includes such tools and offers also in-game debugging possibilities, such as debug drawing, an in-game menu system or console, and the ability to record and replay game flow for testing and debugging purposes.

Nevertheless, most game engines also contain a suite of custom profiling and debugging tools, such as:

- a mechanism for manually instrumenting the code so that specific sections of code can be timed

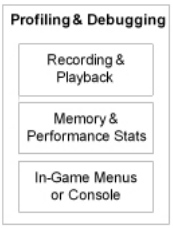


Figure 2.14: Profiling and debugging tools

- a possibility for displaying the profiling statistics on-screen while the game is running
- a way for dumping performance stats to a text file
- an option for determing how much memory is being used by the engine and by each subsystem, including various on-screen displays
- an ability to dump memory usage, high-water mark and leakage stats when the game terminates and/or during gaming
- tools that allow debug print statements and a possibility to turn on or off different categories of debug outputs and control the level of verbosity
- an opportunity to record game events and replay them

2.4.10 Collision and Physics

Collision detection is highly important for every game. Without it, after objects are interpenetrated, it would be impossible to interact with the virtual world in any reasonable way. Some games include a realistic or semi-realistic dynamics simulation which the game industry calls *physics system*. Although the term *rigid body dynamics* is really more appropriated because game developers are usually concerned with the motion (kinematics) of rigid bodies and the forces and torques (dynamics) that cause a motion. This layer is depicted in Figure 2.15.

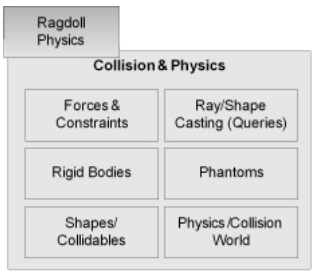


Figure 2.15: Collision and physics subsystem

2.4.11 Animation

A lot of games has organic or semi-organic characters like humans, animals or even robots. For this reason a game engine needs an *animation system* with sprite and texture animation, rigid body hierarchy animation, skeletal animation, vertex animation and morph targets.

Skeletal animation (see Figure 2.16) facilitates a detailed 3D character mesh to be posed by an animator using a relatively simple system of bones. If the bones move, the vertices of the

3D mesh move with them. Although, morph targets and vertex animation are used in some engines, but skeletal animation is the most prevalent animation method in games today.

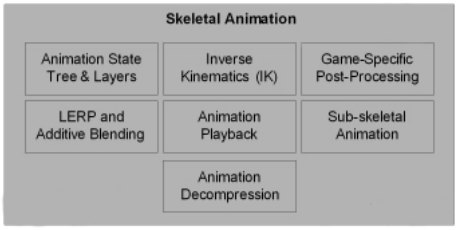


Figure 2.16: Skeletal animation subsystem

Skeletal Mesh Rendering is a component that bridges the gap between the renderer and the animation system (see Figure 2.2). They both cooperate tight together but the interface is well defined. The animation system produces a pose of every bone of the skeleton and afterwards, these poses are passed to the rendering engine as a palette of matrices. The renderer transforms each vertex by the matrix or matrices in the palette in order to generate a final blended vertex position. This process is known as *skinning*.

The animation works closely together with the physics system as well if *rag dolls* takes part of the game. A rag doll is a limp (often dead) animated character whose bodily motion is simulated by the physics system. It determines the positions and orientations of the various parts of the body by treating them as a constrained system of rigid bodies. The animation system calculates the palette of matrices required by the rendering engine in order to draw the character on-screen.

2.4.12 Human Interface Devices (HID)

Human interface devices (HID) describes the input process from the player, including keyboard and mouse, joypad and steering wheels, fishing rods, dance pads or WiiMote.

This component is also referred to a *player I/O* component because the *output* to the player is provided through the HID. A HID layer is shown in Figure 2.17.

The HID engine module is designed to separate the low-level details of the game controller on a particular hardware platform from the high-level game controls. It handles the raw data coming from the hardware, introducing a dead zone around the center point of each joypad stick, de-bouncing button-press inputs, detecting button-down and button-up events, interpreting and smoothing accelerometer inputs and more. Often, it offers a mechanism allowing the player to customize the mapping between physical controls and logical game functions. It sometimes includes a system for detecting chords (multiple buttons pressed together), sequences (buttons pressed in sequence with a certain time limit) and gestures (sequences of inputs from the buttons, sticks, accelerometers and so on).

2.4.13 Audio

Good games contain a good sound design, so that a game engine must also include a *audio engine* (see Figure 2.18) that vary greatly in sophistication. Every game requires a great deal of custom software development, integration work, fine tuning and attention to detail for producing high-quality audio in the final product.

Figure 2.17: Human interface device (HID) layer

Figure 2.18: Audio subsystem

2.4.14 Online Multiplayer/ Networking

Many games permit multiple gamers to play within a single virtual world. Multiplayer games are splitted in four basic types:

- *Single-screen multiplayer*: Two or more human interface devices are connected to a single arcade machine, PC or console. Multiple player characters inhabit a single virtual world. A single camera keeps all player characters in frame simultaneously.
- *Split-screen multiplayer*: Multiple player characters tenant a single world with multiple HIDs connected to a single game machine, but each with its own camera. The screen is divided into sections such that each player is able to view his or her character.
- *Networked multiplayer*: Multiple computers or consoles are networked together where each machine hosts one of the players.
- *Massively multiplayer online games (MMOG)*: A random number of users can play simultaneously within a giant, persistent, online virtual world hosted by a central server.

The layer is depicted in Figure 2.19. The support for multiple players can have a profound impact on the design of certain game engine components. The game world object model, renderer, human input device system, player control system and animation systems are all affected.

Figure 2.19: Online multiplayer subsystem

2.4.15 Gameplay Foundation Systems

Gameplay refers to the actions that takes place in the game, the rules which control the virtual world of the game, the abilities of the player characters (*player mechanics*) and of the other characters and objects in the world as well as the goals and objectives of the player. It is typically implemented either in the native language in which the rest of the engine is written, or in a high-level scripting language. To bridge the gap between the gameplay code and the low-level engine systems, many game engines introduce the *gameplay foundation layer* (see Figure 2.20). This layer provides a suite of core feature, upon which game-specific logics can be implemented conveniently.

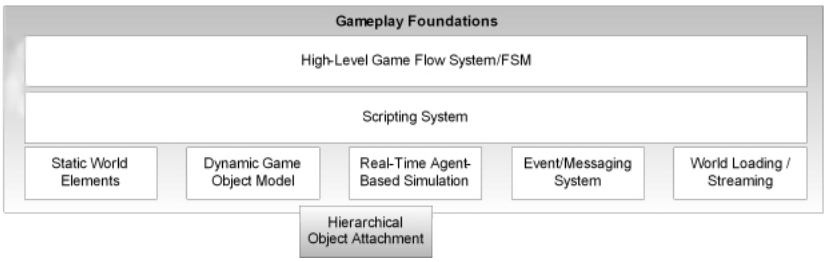


Figure 2.20: Gameplay foundation systems

Game Worlds and Object Models

The contents of a *game world* are usually modeled in an object-oriented manner (but for that not always an object-oriented programming language is used). The collection of object types that complement a game can be called *game object model*. The game object model provides a real-time simulation of a heterogeneous objects collection in the virtual world.

Typical types of game objects are for instance static background geometries (buildings, roads, terrain, etc.), dynamic rigid bodies (rocks, soda cans, chairs, etc.), player characters (PC), non-player characters (NPC), weapons, projectiles, vehicles, lights and cameras.
The *software object models* represent the set of language features, policies and conventions used to implement a piece of object-oriented software.

Event System

Game objects need to communicate with each other. In an event-driven system, the sender creates a small data structure called *event* or *message*, containing the message's type and any argument data that have to be sent. The event is passed to the receiver object by calling its *event handler function*. Events can also be stored in a queue to handle them at some future time.

Scripting System

Most game engines employ a scripting language to support the development of game-specific gameplay rules and content easier and more rapid. Game logic and data can be made by modifying and reloading the script code.

Artificial Intelligence Foundations

Artificial intelligence (AI) was not part of game engines until companies have recognized patterns that arise in almost every AI system. Now it has fallen into the realm of game-specific software.

AI foundation layers can include powerful features, such as:

- a network of path nodes or roaming volumes which define areas or paths where AI characters are free to move without fear of colling with static world geometry
- simplified collision information around the edges of each free-roaming area
- knowledge of entrances and exits of a region and from where in each region an enemy might be able to see and ambush the player
- a path-finding engine based on A* algorithms[13]
- hooks into collision system and world model for line-of-sight (LOS) traces and other perceptions
- a custom world model which tells the AI system where all the entities of interest (friends, enemies, obstacles) are, permits dynamic avoidance of moving objects and so on.

An architecture for the AI decision layer implies the concept of *brains* (one per character), *agents* (each responsible for executing a specific task, such as firing on an enemy or moving from point to point) and *actions* (responsible for allowing the character to perform a fundamental movement which often results in playing animations on the character's skeleton).

2.4.16 Game-Specific Subsystems

On top of the gameplay foundation layer and other low-level engine components, gameplay programmers and designers cooperate to implement features of the game. This layer is depicted in Figure 2.21. Gameplay systems are in general numerous, highly varied and specific to the game being developed. These systems include mechanisms of the player character, various in-game camera systems, artificial intelligence for the control of non-player characters (NPCs), weapon systems, vehicles and many more. At least, some game-specific knowledge seeps down through the gameplay foundations layer and sometimes even extends into the core of the engine itself.

2.5 Asset Pipeline

Games are inherently multimedia applications. All source data (3D models, textures, audio files, etc.) is first developed externally in several tools. Afterwards, they are combined in the game engine. The pipeline from a tool to game engines is often called *asset pipeline*.

[13] computer algorithms for pathfinding and graph traversal, process of plotting an efficiently traversable path between points, called nodes

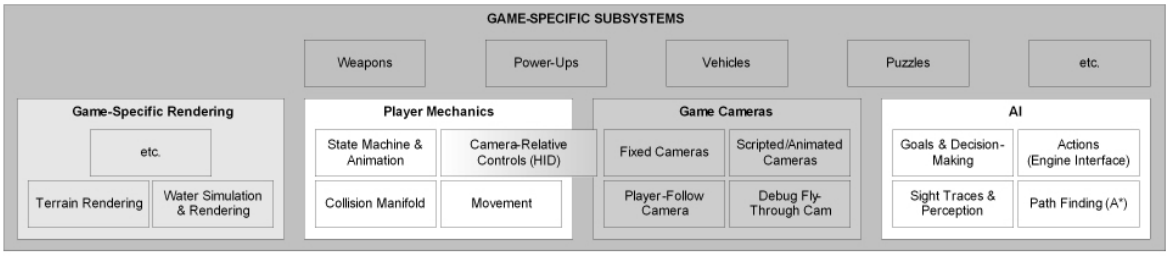


Figure 2.21: Game-specific subsystems

3D Model/ Mesh Data

A *mesh* is a complex shape composed of polygons or triangles and vertices. On today's graphics hardware, all meshes may be translated into triangles before rendering. A mesh usually has one or more materials. They represent visual surface properties, such as color, reflection, bumpiness and more.

Meshes are created in a 3D modeling package such as Maya, 3ds Max or Cinema4D. Exporters must be written to extract the data from the tool and store it on a disk in a format that is compatible with the engine.

Skeleton Animation Data

A *skeletal mesh* is a special kind of mesh. It is bound to a skeletal hierarchy or joint hierarchy for the purposes of articulated animation. Each vertex of such a mesh contains a list of indices indicating to which joint of the skeleton it is bound. A *skin* surrounds the invisible underlying skeleton.

In order to render a skeletal mesh, the game engine requires three distinct kinds of data: the mesh itself, the skeletal hierarchy and one or more animation clips which specify how the joints should move over time.

Audio Data

Audio clips are exported from an audio production tool in a variety of formats and in a number of different data sampling rates. Audio files can be in mono, stereo or other multichannel configurations. They are often organized in databases for the purposes of organization, easy loading into the engine and streaming.

Particle Systems Data

Artist who are specialized in creation of visual effects create complex particle effects for modern games. Many companies develop a custom particle effect editing tool which exposes only the effects that the engine actually supports.

Game World Data

A number of commercial game engines provide good world editors. Writing a good world editor is difficult, but it is an hugely important part of any good game engine.

2.6 Engines for Third-Person Action Adventures

Game engines are typically genre specific. Most genres focus on different technologies[14], so the third-person action adventure also does.

Third-person character-based games have a lot in common with first-person shooters, but with more emphasis on the main character's abilities and locomotion modes. Inter alia, *Ghost Recon*, *Gears of War* and *Uncharted: Drake's Fortune* represent this genre.

Some of the technologies specifically focused on by games in this genre include:

- high-fidelity character animations
- a user input system of detecting complex button and joystick combinations
- accurate hit detection for combats
- moving platforms, ladders, ropes, trellises and other locomotion modes
- puzzle-like environmental elements
- a third-person following camera which always moves with the player character and whose rotation is normally controlled by the player via mouse on a PC or the right joypad stick on consoles
- a complex camera collision system to ensure that the view point never clips through background geometry or dynamic foreground objects

2.7 Scripting languages in Game Development

Nowadays, the probably most commonly used programming language in game development is `C++`. `C++` is a *low-level* language. Low-level language means that they are close to the processor and that is the reason why `C++` is more complex than most other languages and offers usability for speed. For this reason, it is very reasonable for games because nobody wants to develop or play a slow and laggy game since this would slow working time down.

High-level languages such as object-oriented languages like C♯ and Java are further distant from the processor, so it provides speed for usability. But they are not much slower than `C++` so they still make use in game development. Especially, C, C♯ and Phyton are frequently used in game programming too. Java, however, is rarely used.

There are many more scripting languages which are applied in game development, for example JavaScript or Lua. Some game engines even have their own language. For instance, Epic Games implemented its own language called UnrealScript for their popular Unreal Engine.

But it is not about which language is the 'best', it is about which language will support developers to achieve their goal. So it depends on what kind of game have to be developed. The decision about the language does not only depend on these technical aspects. It also depends on the experience and the knowledge of the programmer, so why should an C♯ expert use `C++`? Every game engine provides typically not more then three coding languages.

[14] cp. GREGORY (2009), pp. 13-25.

2.8 Comparison of Game Engines

2.8.1 Criteria

Over the years, a lot of game engines were developed. With the aid of several criteria, the decision to find the best engine can be facilitated. The criteria are divided into *technical* and *non-technical*. Technical criteria embrace the rendering engine, sound engine, physics engine, network module, platforms, scripting languages, input devices, supported formats and features. Non-technical criteria summarize community and support, references, learnability, licensing and costs. These criteria are listed and described in Table 2.1 and 2.2. Every criterion is weighted differently depending on priorities. The tables also contain a value for weighting which tally up to a total of 100.

Criterion	**Description**	**Weighting**
Rendering engine	Embedded graphic SDKs are responsible for the performance of a game.	12
Sound engine	An engine with an embedded sound engine is very useful for the sound design of a game.	12
Physics engine	A good game engine is unimaginable without a physics engine. Almost every game includes physics like collision detection. The physics engine constitutes realistic or semi-realistic simulations.	12
Network module	An existing network module allows to develop online games (browser games, apps, multiplayer mode).	9
Target platforms	Most games are produced for different target platforms in order to increase sales figures.	4
Scripting languages	A game engine supports at least one programming language for the implementation of a game.	5
Input devices	Every engine provides several input devices for video games.	4
Supported formats	Almost all data in a game is created externally in different tools. The more data files or exported standard formats an engine can import, the better.	7
Features	A game engine is characterized by its comprehensive features. This includes features such as rendering possibilities, a system for character animation, incredible visual effects or a powerful audio system.	10
		75

Table 2.1: Technical criteria

Criterion	Description	Weighting
Community and support	The size of the community and support are significant to interested persons. A good community does not only help with trainings, but extends, optimizes and keeps the engine alive. A documentation, that contains introductions, install instructions and solutions for problems, plays a major role in the learnability of a new game engine.	7
References	Existing examples, tutorials and available projects help developers to understand an engine. A good book is even more detailed then a tutorial and shows the coherences and the background of complex issues.	6
Learnability	The community, support and available references improve the learnability value of an engine. The interface and usability are also important aspects that simplify the usage.	8
Licensing and costs	Especially for hobbyartists or smaller companies, the factor of licensing and costs are a big reason for their choice of a specific engine. Many game engines offer test free standard versions and paid premium versions.	4
		25

Table 2.2: Non-technical criteria

2.8.2 Popular Game Engines

Numerous game engines have been developed so far, but not all prevail on the game market. Some famous engines are: CryEngine 3, idTech 5, Source Engine 2, Unity 4 and Unreal Engine 4[15] (see Figure 2.22).

Figure 2.22: Five popular game engines

[15] cp. HOLZBAUER (2010), pp. 1-5.

The german developer studio Crytek released CryEngine with its game *Far Cry* in 2004. The big sellers *Crysis 1-3* were developed in CryEngine 2 and 3. This engine is known for its outstanding graphics quality. Features like volumetric clouds, detailed mimics, plant animation and many more improved the second edition of CryEngine. With its current version, CryEngine 3, Crytek optimizes console games for Xbox 360 and PlayStation 3.

id Software is one of the founders of the shooter genre. It all stared with the games *Doom* and *Quake*. This is why id Tech is also known as Quake-Engine and Doom-Engine. The US developers are famous for their level design, the indoor levels, gloomy corridors and the claustrophobic feeling. But neither *Doom* nor *Quake* represent the fifth version of the engine, id Tech 5. The shooter and racing game mix *Rage* proudly represents id Tech 5 and includes a free open world for the first time inside this engine. With id Tech 5 a legend is back.

In 2004, Valve's Source Engine has already caused quite a stir because of the unexpected high success of *Half-Life* and its popular ego shooter *Counter Strike*. Valve planned two in-house titles based on the Source Engine: *Half-Life 2* and *Team Fortress 2*. Meanwhile, the game studio Troika developed a 'Source-game' called *Vampire: The Masquerade Bloodlines*. Troika licensed this early version of the game and modified it in order to develop a classic RGP with action elements. Therefore, the Source Engine was integrated into the game engines industry with its great graphics and physics engine.

Unity Technologies developed a game engine called Unity 3D which is usable on a wide range of platforms like consoles, mobile devices and web browsers. Games like *WolfQuest* and *Temple Run 2* were developed with this engine. Together with the Unreal Engine, it is the most popular game engine worldwide[16]. In addition, Unity is the most widely used game engine for mobile devices, 53.1 per cent of all developers use Unity[17].

With its Unreal Engine, Epic Games has created a true classic. The long history of this middleware began with the first release in 1998 and the 3D shooter *Unreal*. Epic Games pursued the plan to distribute the engine over a large area, non the less reasonably priced under license. With new hot games, based on the engines that were released on the market, the Unreal Engine increased the level of awareness. In 2002, Epic's Unreal Engine 2 was published. Due to the easy portability to Xbox 360 and PlayStation 3, the Unreal Engine is still tremendously popular. The long list of AAA[18]titles with Unreal Engine 3 includes such illustrious games like *Medal of Honor: Airborne*and *BioShock*. The current version of the engine, Unreal Engine 4, was a co-production between Epic Games and Mozilla, in order to make the engine even running browser based. Thus, it is no exaggeration to say that the Unreal Engine is probably the most popular commercial engine for 3D games and is used by many manufacturers outside the Epic cosmos.

2.8.3 Comparison

Based on the criteria the following five game engines are compared in Table 2.3 und 2.4.

[16] according to a survey conducted by *Game Developer Magazine* (2011)
[17] according to a survey conducted by *Gamasutra* (2012)
[18] AAA is an acronym, each 'A' has a meaning regarding on overall quality: Critical success, innovative gameplay and financial success.

Game engine	**Rendering engine (graphics)***	**Sound engine**	**Physics engine**	**Network module**	**Scripting language**
CryEngine 3	DirectX, AMD Mantle	embedded	embedded	embedded	C++
id Tech 5	DirectX, OpenGL	embedded	embedded	embedded	C++
Source Engine 2	DirectX, OpenGL	embedded	embedded	embedded	C++
Unity 4	DirectX, OpenGL	embedded	embedded	embedded	C♯, Boo, JavaScript
Unreal Engine 4	DirectX, OpenGL, AMD Mantle	embedded	embedded	embedded	UnrealScript

Game engine	**Target platforms**	**Input devices**	**Supported formats**	**Features**
CryEngine 3	Windows, Linux, Xbox 360, Xbox One, PS 3, iOS	keyboard, mouse, joystick	wide range of supported formats	highly complex features
id Tech 5	Windows, Mac OS X, Xbox 360, Xbox One, PS 3, PS 4	keyboard, mouse, joystick	wide range of supported formats	highly complex features
Source Engine 2	Windows, Mac OS X, Linux, PS 3, Xbox 360	keyboard, mouse, joystick	wide range of supported formats	highly complex features
Unity 4	Windows, Mac OS X, Linux, Xbox 360, iOS, PS 3, Wii, Android, Windows Phone 8.x, BlackBerry, Webbrowser, Google Native Client, Flash Player, PS Vita	keyboard, mouse, joystick	extremely wide range of supported formats	complex features
Unreal Engine 4	Windows, Mac OS X, Linux, Dreamcast[19], PS 2, PS 3, Xbox 360, Android, iOS	keyboard, mouse, joystick	wide range of supported formats	highly complex features

Table 2.3: Comparison of technical criteria[20]

[19]Dreamcast is a console of the japanese company Sega. Although the console is no longer available, today still Dreamcast-games are created.

Game engine	Community& Support	References	Learnability	Licensing& costs
CryEngine 3	support, big community, forums (about 600 k posts)	detailed documentation, many tutorials, literature and example projects	okay for beginners, but not suitable best	EULA[21], $9,90 per month, free for non-commercial use
id Tech 5	unknown	no published references	low learnablity in terms of delitescence	no licensing to third-parties
Source Engine 2	support, small community, forums (about 150 k posts)	detailed documentation, just a few tutorials, literature and example projects	relatively difficult learnability without adequate references	proprietary, costs unknown
Unity 4	support, huge community, forums (over 1.5 m posts)	detailed documentation, many tutorials, literature and example projects	good for beginners, easy to learn, a lot of assistance	EULA, Pro version $1.500, free version (with less features)
Unreal Engine 4	support, big community, forums (about 500 k posts)	detailed documentation, many tutorials, literature and example projects	okay for beginners, but not suitable best	EULA, $19 per month, free for non-commercial use (UDK[22])

Table 2.4: Comparison of non-technical criteria[23]

* DirectX, OpenGL and AMD Mantle are the graphic SDKs which are used in game development. DirectX is a collection of APIs for multimedia applications, primarily games, on Windows and Xbox. This SDK usually is an inherent part of every game engine. However, OpenGL supports platform independent programming interfaces. The new revolutionary AMD Mantle has even more performance than DirectX and OpenGL. Games like *Battlefield 4* already support Mantle and there will be many more high-definition games like that in the future.

[20] cp. DBolical Pty Ltd (2014), GameFromScratch (2011) and Wikipedia (2014)
[21] EULA = End User License Agreement
[22] UDK = Unreal Development Kit
[23] cp. Crytek (2014), Valve (2014), Unity Technologies (2014) and Epic Games (2014), numbers of posts are from 24/03/14

2.8.4 Evaluation

Crytek's CryEngine 3 got 91 points in the evaluation (see Table 2.5). It performed best in the technical section. With the graphics SDKs DirectX and AMD Mantle and its amazing rendering features, the quality of CryEngine's graphic is invincible. An embedded sound engine, physics engine and network module are also not missing. The engine uses the most common `C++` as scripting language, which is a big plus. CryEngine 3 distributes its games with the standard input devices (keyboard, mouse, joystick) on a lot of today's most used platforms. With its wide range of supported formats highly complex features can be implemented in order to create a fantastic game. CryDev is Crytek's big community with approximately 600k posts which provide a detailed documentation and support. Many tutorials, literature and example projects help developers to create a game. Due to its high complexity, CryEngine 3 is not suited for beginners but with assistance a lead-in is feasible. This game engine is available for free in non-commercial use. Commercial users have to pay $9,90 per month, which is a really reasonable price for professional developers.

Criterion	**Weighting**
Rendering engine	11
Sound engine	12
Physics engine	12
Network module	9
Target platforms	2
Scripting language	5
Input devices	4
Supported formats	7
Features	10
Community and support	6
References	5
Learnability	5
Licensing and costs	3
	91

Table 2.5: Evaluation of CryEngine 3

The game engine id Tech 5 takes the last place in the ranking (see Table 2.6). It improved highly in contrast to prior versions. DirectX and OpenGL are applied in order to reach a fantastic quality on the graphical level. The embedded sound, physics and network components are also not missing. With its target platforms, input devices and scripting language `C++`, id Tech 5 can easily keep up with other engines. While the engine achieves 69 of 75 points by means of technical criteria, it performed poorly in the non-technical area. Unfortunately, each criteria finally got zero points because there is no information over an existing community, support or any references to documentation, tutorials, literature or example projects. The reason for that id Software does not allow licensing for hobbyartists and third-parties.

Criterion	**Weighting**
Rendering engine	10
Sound engine	12
Physics engine	12
Network module	9
Target platforms	2
Scripting language	5
Input devices	4
Supported formats	6
Features	9
Community and support	0
References	0
Learnability	0
Licensing and costs	0
	69

Table 2.6: Evaluation of id Tech 5

Valve's Source Engine 2 achieves the same results as id Tech 5 in technical department (see Table 2.7). It also uses DirectX and OpenGL as graphic SDKs, has an embedded sound and physics engine and offers the ability to produce online games. The engine provides many features for the mainly demanded target platforms and input devices. With the programming language `C++`, Source Engine 2 allows reputable scripting. But in contrast, the engine is available for third-party studios. The costs are unknown, Valve gives just the declaration of 'extremely competitive prices'. Although there is a small community and support and only a few references can be found on the internet or otherwise. Thereby, the learnability of this game engine is low. Source Engine 2 achieves only a few points in the evaluation in the non-technical section. Overall, it reaches fourth-ranking in the evalution.

Criterion	**Weighting**
Rendering engine	10
Sound engine	12
Physics engine	12
Network module	9
Target platforms	2
Scripting language	5
Input devices	4
Supported formats	6
Features	9
Community and support	3
References	1
Learnability	1
Licensing and costs	1
	75

Table 2.7: Evaluation of Source Engine 2

And the winner is: Unity 4. It is conspicuous that this game engine achieves excellently 94 points (see Table 2.8). The amount of features in Unity is considerably but not as extensive as in other engines. Unity is well-known for its extremely wide range of target platforms. In addition to the standard platforms it is possible to develop a game for Wii or mobile devices like Android, iOS, Windows Phone and BlackBerry. Because of that, Unity is unbeatable in mobile games development. The engine also provides a wide variety of supported industry-standard file formats for 3D modelling, sounds and images. Regrettably, it provides only `C++` only for plugins, not for scripting. The easier languages C♯, JavaScript and Boo are part of the engine. Considering non-technical criteria, Unity 4 stands out compared to the other game engines. A huge community with numerous posts in its forums, stunning support and various references, literature, tutorials and example projects distinguish Unity's high profile. Therefore, an introduction into this engine is easy, especially for beginners. The free version offers lots of features, but of course not all of them. For the whole package, developers need to pay at least $1.500. Mainly due to its demanding applicability and learnablity, Unity 4 deserves to win this evaluation.

Criterion	**Weighting**
Rendering engine	10
Sound engine	12
Physics engine	12
Network module	9
Target platforms	4
Scripting language	4
Input devices	4
Supported formats	7
Features	8
Community and support	7
References	6
Learnability	8
Licensing and costs	3
	94

Table 2.8: Evaluation of Unity 4

Epic Games developed with Unreal Engine 4 an outstanding game engine (see Table 2.9). With the graphic SDKs DirectX, OpenGL and AMD Mantle, rendering takes a major effect at the quality of the game-graphics. Like all other engines, Unreal Engine 4 incorporates a sound engine, physics engine and network module. Games can be developed for the favoured target platforms with different input devices. Epic Games developed also its own scripting language, called UnrealScript, which requires period of vocational adjustment in each case. Sometimes circumstances regarding the import of game data can come up, because the Unreal Engine 4 does not support as many file formats like the other engines. All in all, the evaluation of the technical criteria is greatly successfully. For Unreal developers a big community stands by akin to CryEngine 3: support, numerous forums and lots of references are no question for a triumphant game engine like this one. Though the learnability is maybe out of reach for some amateurs because of the complex features. Epic Games offers the same licensing method as Crytek - free for non-commercial use, paid for commercial use. In conclusion, Unreal Engine 4 hits the third place only one point less than CryEngine 3.

Criterion	**Weighting**
Rendering engine	12
Sound engine	12
Physics engine	12
Network module	9
Target platforms	3
Scripting language	3
Input devices	4
Supported formats	6
Features	10
Community and support	6
References	5
Learnability	5
Licensing and costs	3
	90

Table 2.9: Evaluation of Unreal Engine 4

2.8.5 Conclusion

Without question, all five game engines are impressive inventions and they changed the future of game development forever. The graphics are drop-dead gorgeous and technical features and innovations allow more and more possibilities. The longsellers CryEngine 3, Unreal Engine 4 and Unity 4 eke out a living whereupon CryEngine and Unreal Engine are invincable in the complexibility of their features and Unity is probably suited best for inexperienced users. Source Engine 2 and id Tech 5 are also brilliant engines but not spread out widely for third-parties. No game engine is the best one, they all have advantages and disadvantages. Which game engine should be chosen depends highly on the experience of the developers and on the kind of game that is to be develop as every game and genre has different requirements. Surely it will be exciting to see how these visionary game engines will outclass themselves in the future.

Chapter 3

Patterns in Game Development

In the software development process, architectural design of the software, that have to developed, follows after the requirement definitions. Then, the design of the individual software components ensues. Patterns support developers in both phases. After the design, the actual implementation will take place.

This chapter explains first the definition of software architecture and lists some criteria for an excellent performance.

Afterwards, the question what design and architecture patterns are is clarified. Then, the two identified types of patterns are compared.

A large number of patterns can be used for developing a software. In a final step, three chosen patterns that are typically used for different game objects are described in detail.

3.1 Software Architecture and Patterns

There are a lot of definitions of the term *software architecture*. One of the most commonly used definition is the following one:

> *'The software architecture of a programme or computing system is the structure or the structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.'*[1]

A software architecture describes the externally visible behaviour and properties of the essential components of a software and their relations. It is the connector between demands and the technical realization of a system and is decided on *non-functional requirements*. Non-functional requirements of a software system are: performance, time to market, overall costs of the solution, availability, robustness (fault tolerance), maintainability, convertibility and flexibility, simplicity and so on. So, software architecture is fundamentally for the quality of a software system.

The performance of a software architecture is affected by the following criteria[2]:

1. basis for the implementation of functional and non-functional requirements
2. independence of application specifications
3. clear circumscribed function of components with a high cohesiveness (a component has to solve a logical associated sub problem)
4. explicit definition of the components and their interactions

Before a programmer starts to implement an application, he is dealing with the conception by applying design and architecture patterns.

3.1.1 Architecture Patterns

Architecture patterns describe the software architecture in an abstract way. A software architecture concretizes itsself by the instantiation of an architecture pattern.

Buschmann[3] groups architecture patterns in four categories:

1. *From chaos to structure*: Patterns in this category try to avoid a disorder of components and objects. They support meaningful disassembling of superior tasks into cooperating sub tasks. A famous pattern of this category is the *layer pattern* which deconstructe the application into several layers.
2. *Distributed systems*: This catergory assists the usage of distributed resources and services in networks.
3. *Interactive systems*: This kind of patterns encourages the structuring of interactive software systems. One of the most well-known pattern in the world is the *Model View*

[1] BASS/CLEMENTS/KAZMAN (2013).
[2] SIX/LORENZ/PILGRIM (2007).
[3] BUSCHMANN et al. (2000).

Controller Pattern (MVC). It divides an interactive application into three components: The *model* includes the logic and the data, the *views* represent the data and the *controllers* are responsible for user input.

4. *Adaptive systems*: The patterns of this category stay for the extensions and adaptions (=evolution) of applications.

3.1.2 Design Patterns

Design patterns are approved solutions for recurrent complex design problems. Every pattern describes a special problem and explicate the core of the solution which is often applicable arbitrary. Finally, design pattern are principally solutions for principally problems. They are abstract and include no implementation but instructions.

By studying and applying patterns, a software developer can profit by the pool of other experienced software developers. With the use of patterns the quality of design decisions and the software which will be developed can be improved considerably.

3.1.3 Design Patterns versus Architecture Patterns

The basic difference between design patterns and architecture patterns is the same as between the software conception and architecture conception. The architecture conception shows a software system with its components. The software conception constitutes the single components. Design patterns can be consider as *micro architecture patterns*.

3.2 Patterns in Game Development

Patterns simplify the interaction between objects and components. They are a courtesy for designing the structure of objects, components and architectures. Like the conventional software projects, patterns can also be applied in game development.

3.2.1 Game Architecture Patterns

The *Model View Controller* pattern[4] is suitable in order to reduce the complexibily of games.

Model View Controller

The structure of user interfaces are split into three different roles (separation of concerns) to facilitate the presentation of the same information (see Figure 3.1):

- *View*: presentation of visual elements like windows, buttons, etc.
- *Model*: administration of data independant of their presentation
- *Controller*: control of user input, change of model data, update of visual presentation

With this pattern a substitution of the user interface is possible since the model is independent of the view. Inside the application, the model can be presented by different user

[4] cp. EILEBRECHT/STARKE (2007), pp. 66-68.

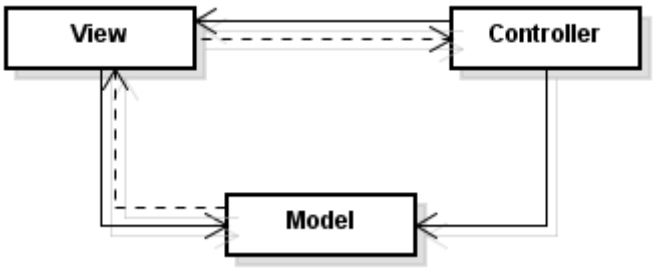


Figure 3.1: Model View Controller pattern

interfaces. Estimations of costs can be better calculated because it is possible to see which one of the three parts is involved.

For instance, a player in a game can be modelled with MVC. The model includes all properties of the player, such as his or her health. If there is any player input, the controller assigns this input to the actions. For example, if the player presses the space bar, the controller interprets this input as a jump and updates the model and the view. The view is responsible for the visual cognition which can include animations or the *Heads-Up Display* (HUD).

3.2.2 Game Design Patterns

Adequated design patterns for developing a game are *Singleton* and *Facade*[5].

Singleton

The design pattern Singleton is an object-based design pattern which makes sure that only one instance of a class can be created. It is easy to implement because only private constructors and static access methods are needed (see Figure 3.2).

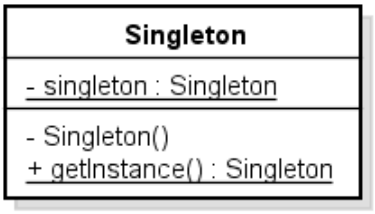


Figure 3.2: Singleton pattern

A well-known example is that there should always exist only one instance of the virtual world in a game and is treated as a global object. *Singleton* makes sure that really only one virtual world is permitted to exist.

[5]cp. BALZERT (2005), pp. 75-79.

Facade

The Facade pattern simplifies the access of a complex subsystem or of a lot of coherent objects. Clients should only know details about this subsystem as little as possible (see Figure 3.3).

For instance, the access to the animations of game characters can result in complex interfaces. With the Facade pattern, it is possible to simplify this access. This concept is very helpful for player characters and enemies.

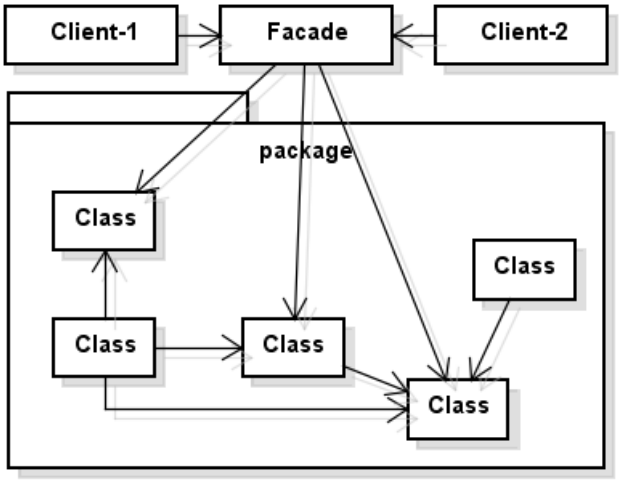


Figure 3.3: Facade pattern

Chapter 4

Unity 3D Features

Although the architecture of game engines is similar, every game engine offers its special unique concepts. So Unity 3D has its own features as well. After the whole content of a game has been created by artists, musicians and other developers using tools and techniques outside of Unity, this game engine makes them work together.

Further, the kinds of options that are made available by the editor vary with editions of Unity. In this chapter, the free version of Unity 4 is described.

First of all, it is important to figure out the user interface of the engine. Unity also contains some basic terms and definitions, which are explained to understand the following context.

Unity even has its own animation system - the so-called *Mecanim*. This system brings human and non-human characters to life with incredibly natural and fluid motion. It is a very new feature that was just released in year 2013 with Unity 4.

Of course, there are also some extras for scripting that allow the implementation special game features. They are listed in the last section of this chapter.

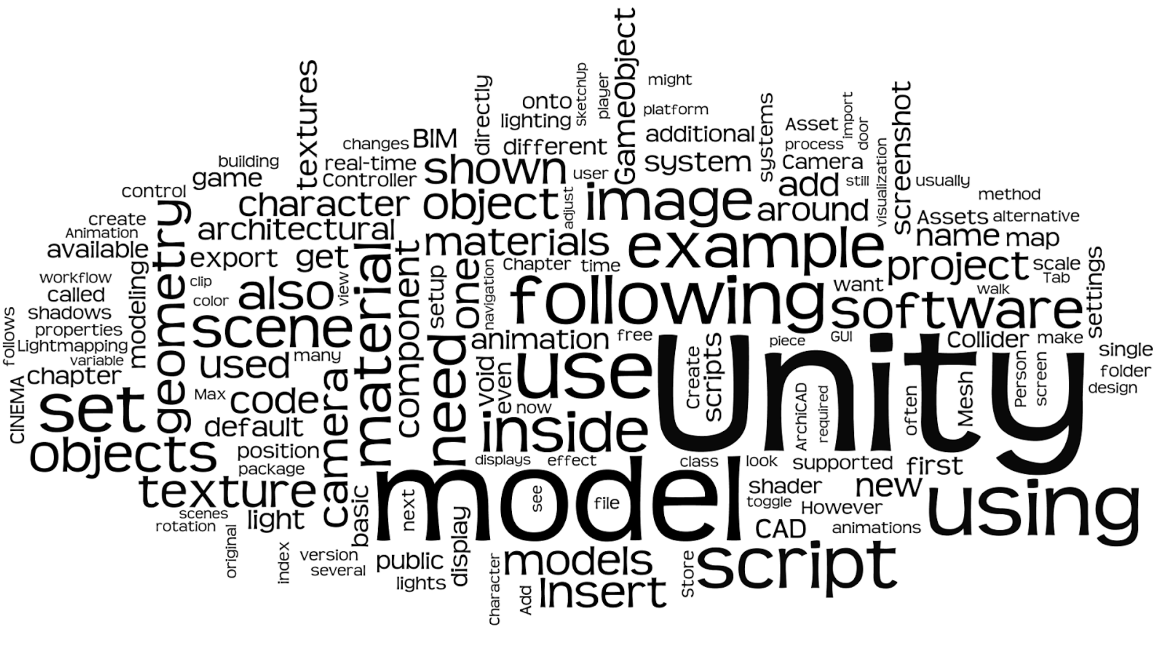


Figure 4.1: A Unity word cloud [1]

[1] `http://cad-3d.blogspot.de/2013/09/my-book-on-unity-for-architectural.html`

4.1 Basic Concepts

4.1.1 The Interface

Unity's interface has a customizable layout consisting of several areas which can be arranged as desired by everyone. Additional windows can be added to the interface. A typical Unity layout is shown in Figure 4.2.

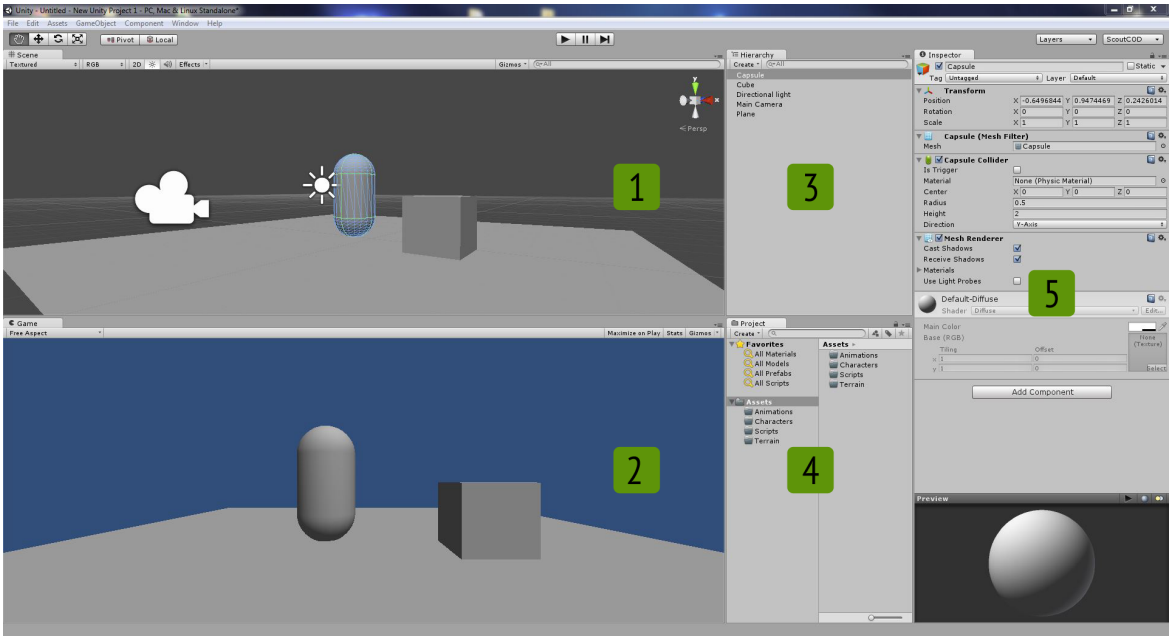


Figure 4.2: Unity's interface and layout

There are five different default panels or views[2]:

1. *Scene*: In the Scene View, the entirety of the game project is build. The window offers a perspective view which is switchable to orthographic views (top-down, side-down, front-on). It acts as a fully rendered editor view of the game world. For the navigation inside this window, Unity supplies a variety of tools.

2. *Game*: A game can be started and tested by clicking the play button. The Game View has also settings for screen resolution. The pause button is very helpful for debugging.

3. *Hierarchy*: The Hierarchy View lists all GameObjects[3]that are placed in the Scene in ascending alphabetical order.

4. *Project*: The Project View is a direct view of the Assets folder of the current project. All files are located there.

5. *Inspector*: The Inspector is like a personal tool kit to customize every element of any GameObject or Asset in a project. By clicking on an object, the Inspector changes to display the relevant properties. Hence, properties of Components can be customized using simple form elements like input boxes or drop-down menus.

[2]cf. MENARD (2011), pp. 16-23.
[3]The following terms of Unity are declared in the next subsection.

There are even more windows that can be placed on the interface. For debugging, the *Console* should always be placed in the layout too. However, animators cannot work without the *Animation* or *Animator* window.

4.1.2 Terminology

It is crucial to be familiar with the definitions and terms[4] of Unity to be able to develope games in 3D.

Assets

Unity projects consists of building blocks (3D models, animations, textures, sound files and so on). In any project there is a folder called *Assets* which includes all files that are used in the game. Unity also offers the *Asset Store* where everyone can download Assets either for free or for money.

Scenes

*Scene*s are comparable to individual levels. If a game includes several levels, every level is designed in a single Scene. It is also advisable to use some Scenes in order to test different parts of the game and save therewith loading time.

GameObjects

Any active object in a Scene is called a *GameObject*. A GameObject initially contains at least one Component, which is the `Transform` Component that represents the position, rotation and scale of an object (described by X, Y, Z coordinates or dimensional in case of scale order). In turn, the Component can be addressed in scripts in order to set an objects position, rotation or scale.

Some examples for GameObjects are:

- *Camera*: A Camera is a device by which players are able to view the world. A Scene can comprise more than one Camera for more view possibilities.
- *Light*: Lights will bring personality and flavour to the game. They illuminate a Scene and objects to create a perfect visual mood. There are four types of Lights: *Directional Light* (affects everything in the *Scene*), *Point Light* (affects objects within its range), *Spot Light* (affects objects within a region, defined by a spot angle and range) and *Area Light* (affects all directions to one side of a rectangular area).
- *Geometric Bodies*: *Cubes*, *Spheres*, *Capsules*, *Cylinders*, *Planes* and *Quads* are the predefined geometric bodies. They are very useful for prototype tests like collision detection.
- *Audio Reverb Zone*: A good game is not imaginable without sound. Audio Reverb Zones are areas with sounds that are activated when a player enters that zone.
- *Terrain*: The Terrain is the world in which the game takes place.

[4] cf. GOLDSTONE (2011), pp. 7-21.

Components

As already mentioned, a GameObject can have more than one *Component*. They can be used for creating behaviour, defining appearance and influencing other aspects of an object's function in the game. To build further interactive elements of a game, scripts can be written. A list of commonly used Components includes:

- *Scripts*: Supported scripting languages of Unity are C♯ (C Sharp), JavaScript and Boo.
- *Rigidbody*: These Components assume control of an object's position. It enables the falling of an object due to the influence of gravity and calculates how objects will act on collision.
- *Character Controller*: A Character Controller is mainly used for third-person or first-person player control that does not make use of Rigidbody physics. The reason to use this Controller instead of a Rigidbody is that the physics of a Rigidbody is not physically realistic. It is simply a capsule-shaped Collider which can be influenced by scripts.
- *Collider*: Colliders are used for detecting a collision between two objects.

Prefabs

Prefabs enable the storing of objects as Assets completely with its Components and the current configuration. Thereby, the object can be reused in different parts of the game and instantiated at any time. If a Prefab is added to a Scene, an instance of it will be created. All Prefab instances are linked to the original Prefab and are essentially clones. The advantage is that if someone makes any changes to the Prefab, the variances are applied to all instances. Another benefit is that this concept is very helpful development in teams. This facilitates team members to export a Prefab in a package and share it within them.

Coordinates

In 3D applications and also in Unity, the information is based on a X, Y, Z format. This is also known as the *Cartesian coordinate* method. All dimensions, rotation values and positions in the 3D world are described in this way.

Local space

All positions of objects in a 3D world are relative to *world zero*. World zero or *origin* represents the position (0,0,0). Developers also use *local spaces* or *object spaces* to define object positions in relation to another object. These relationships are called *parent-child relationships* and will cause that a *child* object moves relative to its *parent* object.

4.2 Colliders and Triggers

4.2.1 Colliders

A *Collider*[5] is an invisible net around an object. Usually, the net simulates its shape and is responsible for reporting any collisions with other Colliders, making the game engine respond accordingly.

Either a Rigidbody Component or a Collider can belong to an object. Rigidbodies are used by physics simulations and Colliders are applied to script customized physics and make the physics of an object more realistic.

Unity offers two main types of Colliders: *Primitives* and *Meshes*. In 3D terms, Primitives shapes are simple geometric objects such as boxes, spheres and capsules. In game development, Primitive Colliders are preferred because they are computationally cheaper or because there is no need for precision. A Mesh Collider is much more expensive. The more complex the mesh, the more detailed and precise the Collider will be and the more expensive it will become. The following diagram illustrates the various types and sub types of Colliders (see Figure 4.3):

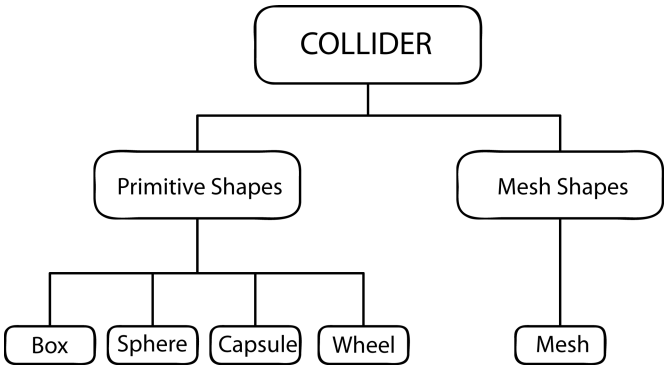


Figure 4.3: Main types of available Colliders

In a game level, most of the objects can be assigned to a Primitive shape. For instance, a bench has a Box Collider, a tent a Sphere Collider or a pillar a Capsule Collider. For a character, developers usually use a *Character Controller* which is similar to a Sphere Collider. Figure 4.4 shows the different Colliders by taking the example of a simple tree in Unity.

There is also a possibility to combine Primitive Colliders. These so-called *Compound Colliders* act as a single Collider and are used for complex meshes for a Mesh Collider can not be applied. Every object should only contain one Collider, so that an object needs to have child objects including a Primitive Collider.

[5] cf. GOLDSTONE (2011), p.16.

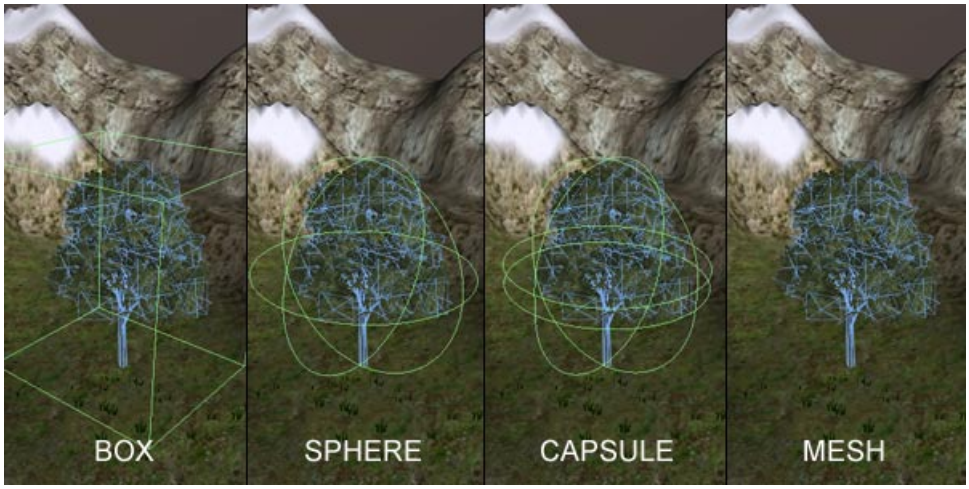


Figure 4.4: Tree with different types of Colliders in Unity[6]

4.2.2 Colliders as Triggers

In addition, Colliders can be set to a so-called *Trigger* mode. In this mode, collisions are still detected but will not be repelled as if they were a physical object. Triggers are often used if a player character is within a particular area. Usually, a Sphere Trigger is favoured because it represents a lifelike area around something. For example, there is a zone around a flame, and if the player enters the Trigger, the flame will light up, otherwise it is lapsed.

4.3 Mecanim Animation Sytem

It is common that characters have several animations that correspond to different actions that can be performed in a game. For its management, Unity has provided the *Legacy animation system*[7].

But with Unity 4.0 the new animation system *Mecanim*[8] was released and offers a lot of advantages compared to the previous system. Mecanim provides a lot of new concepts and terminologies.

- simplified workflow and setup of animations on humanoid characters
- with animation *Retargeting* it is possible to apply animations from one model to another
- preview of animation clips, transitions and interactions
- management of extensive interactions between animations by a visual programming tool
- possibility to animate different body parts

[6] `http://code.tutsplus.com/tutorials/getting-started-with-unity-colliders-unityscript--active-8367`
[7] The Legacy animation is still available, but it is not recommended to use it for new projects.
[8] Unity Technologies (2014).

4.3.1 Humanoid Characters

A rigged and skinned *humanoid* model (see Figure 4.5) is required in order to take full advantages of Mecanim's humanoid animation system and its Retargeting.

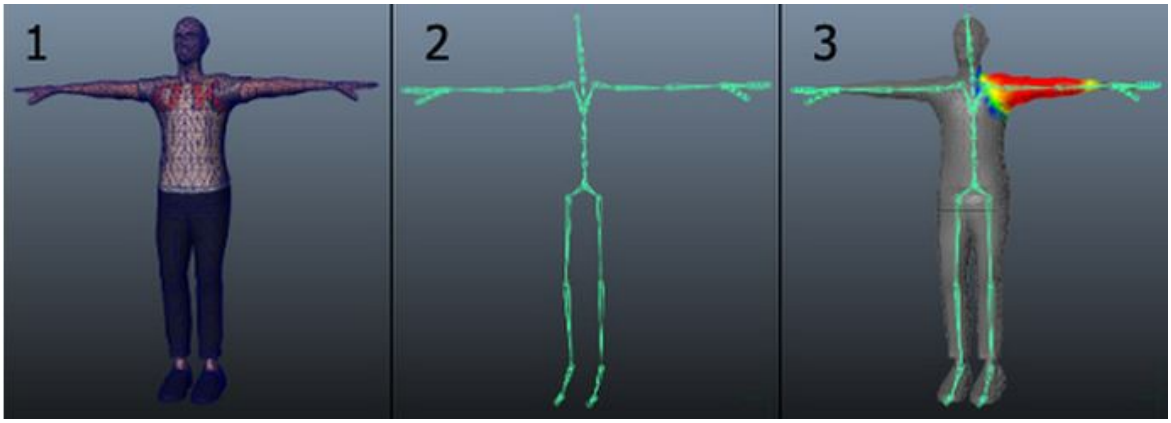


Figure 4.5: Stages for preparing a character (modelling, rigging, skinning) [9]

1. *Modelling*: A character model is created in a 3D modelling package like 3DSMAX, Maya, Blender or the like. In general, it consists of polygons or is converted into a triangulated mesh and should be modelled in T-Pose because this enables refinement of polygon details and simplifies the positioning of the rig inside the mesh.

2. *Rigging*: The mesh has to contain a joint hierarchy or skeleton that defines the bones inside the mesh and their movement in relation to each other. Every animated mesh needs such a rig.

3. *Skinning*: The model has to be connected to the skeleton because this step defines the parts of a mesh that move when a given joint is animated. A common method is to assign individual vertices and the weighting of influence per bone onto the mesh.

Unity provides the import of native Maya and Cinema4D files but also generic FBX[10]files that can be exported from most animation packages. After a character model file is imported, humanoid properties can be specified as rig type. After that, an *Avatar* gets created. The Avatar is a fundamental aspect of the Mecanim System and represents the joints hierarchy of a mesh. It is important that this will be configured according to the model.

Non-humanoid animations are also supported despite the use of an Avatar system or other features. In Mecanim terminology, non-humanoid animations are referred to *Generic Animations*. The skeleton of such meshes can be arbitrary, but the referenced bones still need to be specified.

[9] `http://docs.unity3d.com/Documentation/Manual/UsingHumanoidChars.html`
[10] FBX (Filmbox) is a file format that is used to provide compatibility between digital content creation applications.

4.3.2 Animation State Machines

A character in a game has several animations. Usually, the player is able to control different locomotions, like for example, swimming by entering a lake or he starts to dancing in idle states. The management of all animations is a complex scripting task. Mecanim offers a computer science concept, also known as *State Machine*, that simplifies the management and sequencing of a character's animation.

Animator and Animator Controller

An *Animator Controller* is provided by Unity and enables to maintain a set of animations of a character and to switch between them on certain game conditions. The Animator Controller manages the transitions between animations using a State Machine, a kind of a simple programme, written in a visual programming language within Unity. The Avatar and Animator Controller are finally applied to an object by attaching an Animator Component that references them.

One of the most powerful features of Mecanim is *Retargeting*. This means that the same set of animations is applied to various character models. Retargeting is only possible for humanoid models for which an Avatar has been configured because this serves as correspondence between the models' bone structure.

Basics

The basic idea is that a character is engaged in some particular kinds of actions at any given time. Every animation like walk, run, jump, and so on, is called a *state*. The character will have some specifications to switch to the next state. The options for the next state that a character can enter from its current state are related by state *transitions*.

The states and transitions of a State Machine can be figured as a graph diagram (see Figure 4.6) where nodes represent the states and arrows between them the transitions. The current state behaves like a marker that is placed on one of the nodes and can then only jump to another node along one of the arrows.

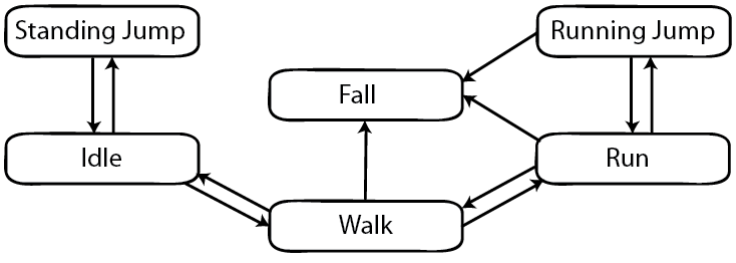


Figure 4.6: State Machine basics

According to the example, a player can switch from *Idle* state to *Standing Jump* or *Walk* according to the player input, for example, if the player presses 'space'. The player can always switch to another state if there exist a transition. For instance, it is not possible to go from *Idle* to *Fall* because there is no connection between these states. The player needs to be in *Walk*, *Run* or *Running Jump* to reach *Fall*. Due to the fact that there are only transitions to the state *Fall* but not back, the result is that the game will be over in that state.

This system allows animators to work more independently from programmers. State Machines can be designed and updated quite easily. Each state has a motion associated with it that will be played whenever the machine is in that state. It is possible to change parameters during tests to see if the animations work. Animation State Machines can be set up in the Animator Controller window (see Figure 4.7). They consist of states, transitions, events and even smaller Sub-State Machines used as Components in larger machines.

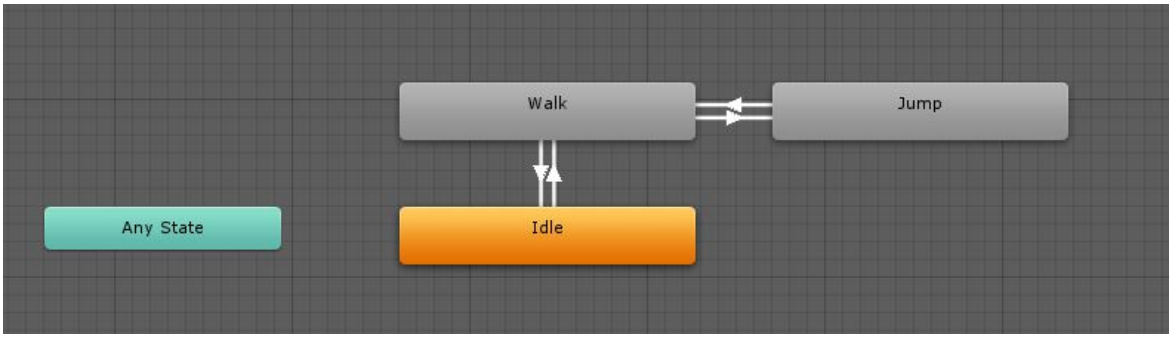


Figure 4.7: A State Machine in Unity

States

Animation State Machines consist of animation states which are the basic building blocks. Each state contains an individual animation sequence that is played during the character is in that state. If an event in the game triggers a state transition, the character will be switched in a new state and its animation sequence will be adapted. States can also contain Blend Trees[11].

The default state, highlighted in orange, is the initial state of a State Machine. Usually, this is an idle animation of the character. There is also a box, highlighted in turquoise. This so-called *Any State* is a special state which is always present. This is a shortened way to add the same outgoing transition to all other states in a Machine. For example, a character can do an exhaust animation from every state like jogging, jumping or something else.

[11]The definition of Blend Trees follows on the next page.

Transitions

Animation *transitions* define how an object can switch from one state to another. A condition consists of:

- an event parameter or *Exit Time*, which specifies a number and represents the normalized time of the source state (for example 0.95 means the transition will be triggered after 95 per cent of the clip have been played)
- a conditional predicate, if needed (for example, *less/ greater* for `float` parameters or *true/ false* for `bool` variables)

Parameters

Animation *parameters* are variables that are defined within the animation system and control the transitions between states. A programmer can assign them in scripts.

The default parameter values can be added and initialized in the Animator window. Unity borrows five types of parameters:

- A `vector` parameter defines a point in world.
- An `integer` parameter represents a total number.
- A `float` parameter is a floating-point number.
- A `boolean` parameter can be true or false.
- A `boolean` parameter is reset if another state is entered.

Blend Trees

Blend Trees are a special state type in an Animation State Machine. With them, multiple motions can be blended smoothly to all varying degrees. The animations are controlled by using a specified blending parameter. This parameter must be a `float` variable. If the first animation has an value of 0 and the second one a value of 1, the *State Machine* computes the values between 0 and 1 for the blending effect.

A Blend Tree is often used for a locomotion state which can be controlled by the player in different directions. So the blending parameter is *Direction* and the motions can be *WalkLeft*, *WalkRight* and *WalkForward*. Unity provides a graphical visualization of how the motions are combined if the parameter value changes (see Figure 4.8).

There are two options of setting up a *Blend Tree*:

1. *1D Blending*: This Blend Tree is the default blend type and is controlled by only one parameter.
2. *2D Blending*: 2D Blend Trees are controlled by two blending parameters. There are three different types of 2D Blending. They differ in how the influence of each motion is calculated:

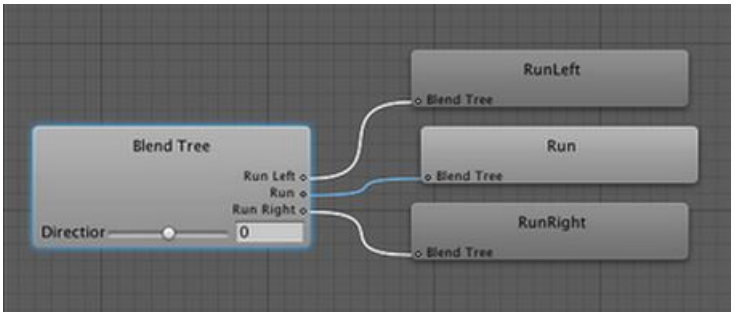


Figure 4.8: Blend Tree

- *2D Simple Directional*: The animations represent different directions (for example *WalkForward, WalkBackward, WalkRight, WalkLeft*). Optionally a single motion at position (0,0) can be included, such as *Idle*. The Simple Directional type does not include multiple motions in the same direction, such as *WalkBackwards* and *RunBackwards*.
- *2D Freeform Directional*: The motions represent different directions. This blend type can contain multiple motions in the same direction like *WalkBackwards* and *RunBackwards*. The set of animations includes a single motion at position (0, 0) such as *Idle*.
- *2D Freeform Cartesian*: This blend type is used if the set of animations does not represent different directions. With Freeform Cartesian the `x` and `y` parameter can define different concepts, such as angular and linear speed. Motions like *WalkForwardNoTurn*, *RunForwardNoTurn*, *WalkForwardTurnRight* or *RunForwardTurnRight* are characteristical for this type.

Sub-State Machines

The more states and transitions a State Machine contains, the more confusing the system gets. *Sub-State Machines* produce a better management inside the State Machine. If there are some states which can be summarized, it is advisable to create a Sub-State Machine which is connected to the *Base Layer*. For instance, a character may have five several idle animations. They can be combined visually to a group instead of five single states. Sub-State Machines do not change the functionality, they are an instrument for better organization of the states.

4.4 Scripting Reference

Unity provides an amount of useful classes and methods in order to implement a game[12].

4.4.1 The MonoBehaviour class

The class `MonoBehaviour` extends the class `Behaviour` which specializes the class `Component` that finally originates from the class `Object`. So `Object` is the base class for all referenced objects in Unity. `Component` is the base class for everything attached to `GameObjects` which is the base class for all entities in Unity *Scenes*. `Behaviours` are `Components` that can be enabled or disabled and at least, `MonoBehaviour` is the base class every script derives from. Figure 4.9 shows the corresponding class diagram:

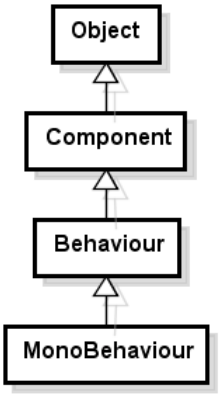


Figure 4.9: Class diagram of MonoBehaviour

4.4.2 Life Cycle of a Script

A Unity script contains messages[13]that define the life cycle of a script (see Listing 4.1):

```
1  public class MonoBehaviour : Behaviour
2  {
3    ...
4    // Startup
5    void Awake();
6    void OnEnable();
7    void Start();
8    ...
9    // Updating
10   void FixedUpdate();
11   void Update();
12   void LateUpdate();
13   ...
14   // Teardown
15   void OnDisable();
16   void OnDestroy();
17   ...
18 }
```

Listing 4.1: MonoBehaviour.cs

[12] UNITY TECHNOLOGIES (2014).
[13] In Unity, methods are also called functions, callback-methods are called messages.

Figure 4.10 shows exactly when a bunch of the most important things happens.

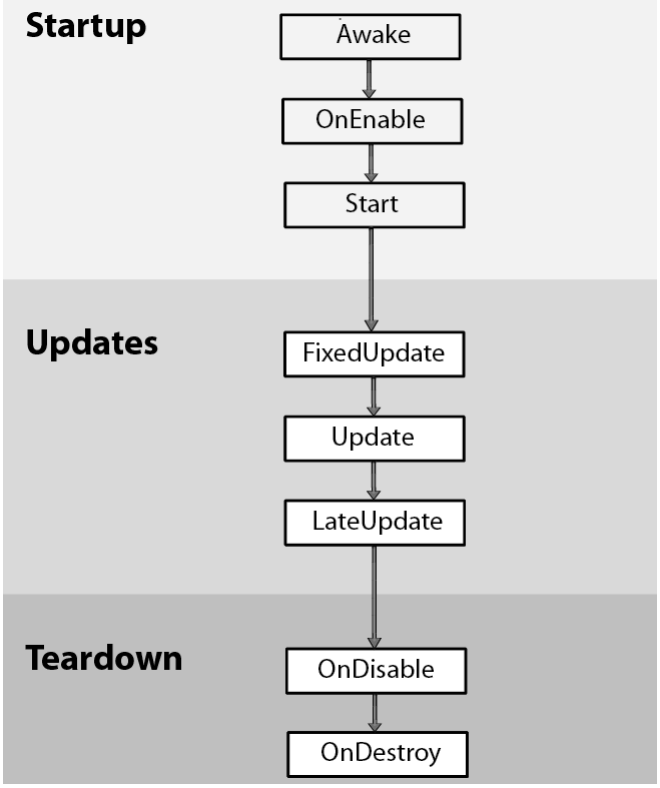


Figure 4.10: MonoBehaviour life cycle

Startup

`Awake()` and `Start()` are two messages that are called during a script is loaded. `Awake()` is called first, even if the script `Component` is not enabled. It is used for initialize any references between scripts. `Start()` is called after `Awake()` and immediately before the first `Update()`, but only if the script `Component` is enabled. `Awake()` and `Start()` are both only called once in the lifetime of a script that is assigned to `GameObject`. In contrast, the method `OnEnable()` is called whenever the object becomes enabled.

Updates

`Update()` is called every frame. This message is not called on a regular timeline, its interval times vary. `FixedUpdate()` is a similar message like `Update()` but it has a fundamental difference. It is called on a regular timeline, the intervals are consistent. `LateUpdate()` is called once per frame, after `FixedUpdate()` and `Update()` have finished. Any calculations that are performed in `Update()` will have completed when `LateUpdate()` begins. A common use of `LateUpdate()` would be a third-person camera. This will ensure that the character has moved completely before the camera tracks its position.

Teardown

There are some messages to teardown a script: `OnDisable()` is called if the script becomes disabled or inactive and `OnDestroy()` is called if the `MonoBehaviour` will be destroyed.

4.4.3 Input

Some valuable methods of Unity's `Input` class are enumerated in Listing 4.2 that are able to receive player input from keyboard, mouse and joypad.

```
1  public class Input
2  {
3    ...
4    // GetKeyXXX()
5    static bool GetKey(string keyName);
6    static bool GetKey(Keycode keycode);
7    static bool GetKeyDown(string keyName);
8    static bool GetKeyDown(Keycode keycode);
9    static bool GetKeyUp(string keyName);
10   static bool GetKeyUp(Keycode keycode);
11
12   // GetButtonXXX()
13   static bool GetButton(string buttonName);
14   static bool GetButtonDown(string buttonName);
15   static bool GetButtonUp(string buttonName);
16
17   // GetMouseButtonXXX()
18   static bool GetMouseButton(int mouseButton);
19   static bool GetMouseButtonDown(int mouseButton);
20   static bool GetMouseButtonUp(int mouseButton);
21
22   // GetAxis()
23   static float GetAxis(string axisName);
24   ...
25 }
```

Listing 4.2: Input.cs

GetButtonXXX(), GetKeyXXX() and GetMouseButtonXXX()

`GetButton()` uses keys that can be specified in the *Input Manager* of Unity. The Input Manager is where all the different input axes and buttons are defined for a project. This facilitates the reusability of specific keys. If the player presses a button, `GetButtonDown()` is called for one single frame. `GetButton()` is called until the player releases the button. Afterwards, `GetButton()` returns to false and `GetButtonUp()` is true, but also for just one single frame. The methods always return a `bool`.

`GetKey()`, `GetKeyDown()` and `GetKeyUp()` behaves in a similar way. The only difference is that instead of the specified button names, key names or keycodes are used.

`GetMouseButton()`, `GetMouseButtonDown()` and `GetMouseButtonUp()` are used for mouse buttons. The particular key is represented with an `integer` value.

GetAxis()

The method `GetAxis()` has some fundamental differences. It returns a `float` value between -1 and 1. The virtual axes are also specified in the Input Manager. By default, the horizontal axis uses 'A' and 'D' or the arrows 'left' and 'right'. The vertical axis uses the keys 'S' and 'W' or the arrows 'up' and 'down'.

Figure 4.11 depicts an example of the input of horizontal and vertical axes with the keys 'WASD'. If the player presses 'W' (v = 1) he moves forwards, with 'S' backwards, with 'D' to the right and with 'A' to the left. If the player presses 'S', releases this key and starts to press another key like 'W' and 'A' together, `GetAxis()` computes the intermediate values in `floats` within a few milliseconds. The red quad represents the current input and the returned values.

(a) Input.GetAxis() values [14] (b) Imagination of visual axes

Figure 4.11: Behaviour of GetAxis()

Scripting example:

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class GUIExample : MonoBehaviour
5  {
6    float v = Input.GetAxis("Vertical");
7    ...
8    if(Input.GetKeyDown("q")
9      print("The key 'Q' was pressed.");
10
11   else if(Input.GetButton("Jump"))
12     print("The button 'Jump' (space) is hold.");
13
14   else if(Input.GetMouseButtonUp(1))
15     print("The left mouse button was released.");
16
17   else if(v > 0)
18     print("The player is moving forwards");
19 }
```

Listing 4.3: InputExample.cs

In Listing 4.3, the variable `v` represents a `float` value of the vertical axis. `v > 0` means that the player presses by default 'W' and walks forward. If he or she would want to walk backwards `v` would be `< 0` (l. 6, ll. 17-18). The commands pressing key 'Q', button 'Jump' or the left mouse button evoke a simple message (ll. 8-15).

[14] https://unity3d.com/learn/tutorials/modules/beginner/scripting/get-axis

4.4.4 Animator

`OnAnimatorMove()` is a message for processing animation movements for modifying root motion. This callback will be called at each frame after the State Machines and the animations have been evaluated (see Listing 4.4).

```
1 class MonoBehaviour : Behaviour
2 {
3     ...
4     void OnAnimatorMove();
5     ...
6 }
```

Listing 4.4: MonoBehaviourOnAnimatorMove.cs

Scripting example:

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class AnimatorExample : MonoBehaviour
5  {
6    private Animator animator;
7
8    void Start()
9    {
10     animator = GetComponent<Animator>();
11   }
12
13   void OnAnimatorMove()
14   {
15     if(animator)
16     {
17       Vector3 newPosition = transform.position;
18       newPosition.z += animator.GetFloat("Speed") * Time.deltaTime;
19       transform.position = newPosition;
20     }
21   }
22 }
```

Listing 4.5: AnimatorExample.cs

In this instance (see Listing 4.5), the new position of the character is computed by means of the *Animator Controller* (ll. 17-19). It depends on the `float` parameter 'Speed' which is influenced by player input. `Time.deltaTime` is the time in seconds it took to complete the last frame. It is used to make the game framerate independent. So, the character is moving constantly per second and not per frame.

The class `Animator` provides some necessary getter and setter methods in order to work with Mecanim. They are enumerated in Listing 4.6. The methods access the parameters via their names inside the State Machine. Depending on the data type, the parameter can be requested with one of the setter methods (ll. 8-10) with the indication of the parameter name. Parameters can also can also be changed via setter methods with the additionally declaration of the new values (ll. 13-17). Transitions in State Machines can respond and be influenced thereby.

```
1  public class Animator : Behaviour
2  {
3    ...
4    // getter methods
5    bool GetBool(string name);
6    float GetFloat(string name);
7    int GetInteger(string name);
8    ...
9    // setter methods
10   void SetBool(string name, bool value);
11   void SetFloat(string name, float value);
12   void SetInteger(string name, int value);
13   void SetTrigger(string name);
14   void SetTrigger(int id);
15   ...
16 }
```

Listing 4.6: Animator.cs

4.4.5 Random

The class `Random` provides the method `Range()` (see Listing 4.7). As the name implies, with this method it is possible to implement new methods that behave randomly.

```
1  public class Random
2  {
3    ...
4    static float Range(float min, float max);
5    static int Range(int min, int max);
6    ...
7  }
```

Listing 4.7: Random.cs

Listing 4.8 shows an example in which five different audio clips are enabled (l. 10). The sounds are randomly picked (l. 15) and played (l. 16).

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class RandomExample() : MonoBehaviour
5  {
6    public AudioClip[] sounds;
7
8    void Start()
9    {
10     sounds = new AudioClip[5];
11   }
12
13   void Update()
14   {
15     audio.clip=sounds[Random.Range(0, 4)];
16     audio.Play();
17   }
18 }
```

Listing 4.8: RandomExample.cs

4.4.6 Collision and Trigger Detection

`MonoBehaviour` contains useful messages for detecting collisions (see Listing 4.9).

```
1  public class MonoBehaviour : Behaviour
2  {
3    ...
4    // collision detection
5    void OnCollisionEnter(Collision collision);
6    void OnCollisionExit(Collision collision);
7    void OnCollisionStay(Collision collision);
8
9    // trigger detection
10   void OnTriggerEnter(Collider other);
11   void OnTriggerExit(Collider other);
12   void OnTriggerStay(Collider other);
13
14   // controller detection
15   void OnControllerColliderHit(ControllerColliderHit hit);
16   ...
17 }
```

Listing 4.9: MonoBehaviourCollisions.cs

- `OnCollisionEnter()` is called when the Collider has begun touching another Collider. `OnCollisionEnter()` does not work with Character Controller Colliders.
- `OnCollisionExit()` is called when the Collider has stopped touching another one.
- `OnCollisionStay()` is called once per frame for every Collider that is touching another Collider.
- `OnControllerColliderHit()` is called when the Controller hits a Collider while performing a move. This can be used to push objects when they collide with the character.
- `OnTriggerEnter()` is called when the Collider enters the Trigger.
- `OnTriggerExit()` is called when the Collider has stopped touching the Trigger.
- `OnTriggerStay()` is called almost all frames for every Collider that is touching the Trigger.

Scripting example:

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class CollisionExample : MonoBehaviour
5  {
6    void OnCollisionEnter(Collision col)
7    {
8      if(col.gameObject.name == "Box")
9        print("Our object collided with the box");
10   }
11 }
```

Listing 4.10: CollisionExample.cs

The message in Listing 4.10 needs a `Collision` parameter which describes the collision. If the player touches something with a Collider it checks if the hit object has the name 'Box'. If it is actual this box it prints 'Our object collided with the box', else the message do not do anything.

4.4.7 GUI and HUD

Any GUI element must exist within Unity's `OnGUI()`[15] message (see Listing 4.11). It is called for rendering and handling GUI events. This means that `OnGUI()` might is called several times per frame and renders.

It is also possible to create a 2D *Heads Up Display (HUD)* with this message. A HUD in video games varies between differing genres. While a racing game needs elements like speed, position and so on, a shooter is made up of elements like health, ammunition and inventory items. The HUD is always controlled by a script containing different textures and text.

```
1 class Monobehaviour : Behaviour
2 {
3     ...
4     void OnGUI();
5     ...
6 }
```

Listing 4.11: MonoBehaviourOnGUI.cs

Scripting example:

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class GUIExample : MonoBehaviour
5  {
6    public Texture2D healthBar;
7    private int health = 100;
8    ...
9    void OnGUI()
10   {
11     GUI.DrawTexture(new Rect(10, 10, health, 50), healthBar);
12   }
13 }
```

Listing 4.12: GUIExample.cs

This code (see Listing 4.12) simply draws a health bar onto the HUD (l. 11). The texture can be assigned in the Inspector. This is why this variable needs to be `public` (l. 6). `Rect` is defined by the x, y position and width, height. Here, the width is the variable health. That means that the width of the health bar changes if the health variable changes so if the player looses or regenerates his health.

[15] cf. MENARD (2011), pp. 309-318.

4.4.8 Coroutines

`MonoBehaviour` also contains another helpful type of methods, called `Coroutine`[16] is like a method that has the ability to pause execution and return control to Unity but then to continue where it left off on the following frame or after the indicated time. Coroutines are executed after all `Update()` messages (see Listing 4.13).

```
1 class Monobehaviour : Behaviour
2 {
3   ...
4   Coroutine StartCoroutine(IEnumerator routine);
5   Coroutine StartCoroutine(string methodName, object value = null);
6   void StopCoroutine(string methodName);
7   void StopAllCoroutines();
8   ...
9 }
```

Listing 4.13: MonoBehaviourCoroutines.cs

Scripting example:

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class CoroutineExample : MonoBehaviour
5  {
6    void Update()
7    {
8      if(Input.GetKeyDown("c"))
9        StartCoroutine("DoSomething");
10
11     else if(Input.GetKeyDown("d"))
12       StopCoroutine("DoSomething");
13
14     else if(Input.GetKeyDown("f"))
15       StartCoroutine("DoSomethingElse");
16
17     else if(Input.GetKeyDown(KeyCode.Escape))
18       StopAllCoroutines();
19   }
20
21   IEnumerator DoSomething()
22   {
23     yield;
24     print("Do something");
25   }
26
27   IEnumerator DoSomethingElse()
28   {
29     yield return new WaitForSeconds(2);
30     print("Do something");
31   }
32 }
```

Listing 4.14: CoroutineExample.cs

[16] cf. THORN (2013), pp. 159-162.

To set a Coroutine running, the `StartCoroutine()` method is used. The parameter is the name of the Coroutine. In this example (see Listing 4.14), the Coroutine starts `DoSomething()` if the player presses the key 'C' (ll. 8-9). If he presses 'F' (ll. 14-15), the second Coroutine starts. It can be stopped with `StopCoroutine()`, in this case `DoSomething()` can be stopped with pressing 'D' (ll. 11-12). All Coroutines running inside the script can be paused with `StopAllCoroutines()` (ll. 17-18).

Coroutines are only applicable to methods that are declared with a return type of `IEnumerator` and with the `yield return` statement included somewhere in the body. The `yield return` line is the point at which execution will pause and be resumed the following frame (`DoSomething()` (ll. 21-25)). It is also possible to introduce a time delay using `WaitForSeconds` like `DoSomethingElse()` includes. In the example, the Coroutine `DoSomething()` is paused for two seconds (ll. 27-31). Afterwards, it continues.

4.4.9 Invoke

`Invoke()` and `InvokeRepeating()` allows developers to schedule method calls to occur at a later time. But only methods with no parameters and a return type of `void` can be called using `Invoke()`. The methods are enumerated in Listing 4.15.

```
1  class Monobehaviour : Behaviour
2  {
3    ...
4    // invokes
5    void Invoke(string methodName, float time);
6    void InvokeRepeating(string methodName, float time, float repeatRate);
7    bool IsInvoking(string methodName);
8    void CancelInvoke();
9    void CancelInvoke(string methodName);
10   ...
11 }
```

Listing 4.15: MonoBehaviourInvoke.cs

Scripting example:

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class InvokeExample : MonoBehaviour
5  {
6    void Start()
7    {
8      Invoke("DoSomething", 2);
9      InvokeRepeating("DoSomethingRepeated", 2, 5);
10   }
11
12   void Update()
13   {
14     if(IsInvoking("DoSomethingRepeated"))
15       CancelInvoke("DoSomethingRepeated");
16
17     if(Input.GetKeyDown("r")
18       CancelInvoke();
19   }
```

```
20
21    void DoSomething()
22    {
23        print("Do Something!");
24    }
25
26    void DoSomethingRepeated()
27    {
28        print("Repeat!");
29    }
30 }
```

Listing 4.16: InvokeExample.cs

Listing 4.16: `Invoke()` takes two parameters. The first one is a `string` for the name of the method we wish to call and the second is the time when this method shall be invoked in seconds. So in this example, the method `DoSomething()` is called after two seconds for a single time (l. 8).

Many tasks in a game need to be carried out periodically. `InvokeRepeating()` has three parameters - the same as `Invoke()` plus an argument for the time interval. It calles the method `DoSomethingRepeated()` after two seconds and repeat the invoke every five seconds until it is stopped or the `MonoBehaviour` is teared down (l. 9).

In order to stop all instances of an `Invoke()` call, the method `CancelInvoke()` is used. The method `DoSomethingRepeated()` is cancelled when any invoke is pending on the method `DoSomethingRepeated()` (ll. 14-15). If someone presses 'R', all invokes are cancelled (ll. 17-18).

4.4.10 Transform

The `Transform` class which is descended from `Component` describes the position, rotation and scale of an object (see Figure 4.12). Every object in a scene has a *Transform* component that is used to store and manipulate theses properties. Every `Transform` object can have a parent, which allows to apply position, rotation and scale hierarchically.

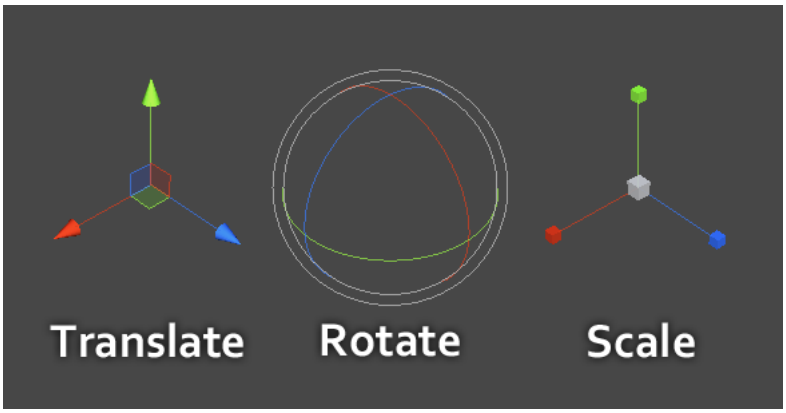


Figure 4.12: Properties of Transform [17]

[17] `http://3dgep.com/?p=3246`

```
1  class Transform : Component
2  {
3    ...
4    Vector3 TransformPoint(Vector3 position);
5    Vector3 TransformPoint(float x, float y, float z);
6    Vector3 InverseTransformPoint(Vector3 position);
7    Vector3 InverseTransformPoint(float x, float y, float z);
8
9    Vector3 TransformDirection(Vector3 direction);
10   Vector3 TransformDirection(float x, float y, float z);
11   Vector3 InverseTransformDirection(Vector3 direction);
12   Vector3 InverseTransformDirection(float x, float y, float z);
13
14   void Rotate(Vector3 eulerAngles, Space relativeTo = Space.Self);
15   void Rotate(float xAngle, float yAngle, float zAngle, Space relativeTo
       = Space.Self );
16   void Rotate(Vector3 axis, float angle, Space relativeTo = Space.Self);
17
18   void LookAt(Transform target, Vector3 = Vector3.up worldUp);
19   void LookAt(Vector3 worldPosition, Vector3 = Vector3.up worldUp);
20   ...
21 }
```

Listing 4.17: Transform.cs

Some important methods are registered in Listing 4.17:

- `TransformPoint()` (ll. 4-5) transforms its parameter `position` (or position `x`, `y`, `z`) from local space to world space. The method `InverseTransformPoint()` (ll. 6-7) is the opposite and transforms `position` from world space to local space. The returned position is affected by scale.

- `TransformDirection()` and `InverseTransformDirection()` (ll. 9-12) behave analog but instead of the parameter `position`, `direction` is used. These methods are used for dealing with directions. The returned position is not affected by scale.

- `Rotate()` (ll. 14-16) applies a rotation around the y axis. The first method takes two parameters: `eulerAngles` and `relativeTo`. If the second parameter is left out, the rotation is applied around the transform's local axes. `Rotate()` can also contain four other parameters(`xAngle`, `yAngle`, `zAngle` and again `relativeTo`. Now a rotation of the three degrees around the three axes is applied. A third concept is to rotate the transform around an axis by angle degrees. For this goal three parameters are required: `axis`, `angle` and `relativeTo` (can be left again). If `relativeTo` is set to `Space.World`, the `axis` parameter is relative to the world x, y, z axes.

- `LookAt()` (ll. 18-19) takes two parameters. `target` represents the object to point towards and `worldUp` is a vector that specifies the upward direction. The method rotates the transform so the forward vector points at the current position of the target. Then it rotates the transform to point its up direction hinted at by the `worldUp` vector. This parameter can also be left out, in this case, the method will use the world y axis. Instead of the `target` parameter, `LookAt()` can also contain the `worldPosition`. So

now the method rotates the transform so the forward vector points at`worldPosition` which is a point in the world to look at.

Scripting example:

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class TransformExample : MonoBehaviour
5  {
6    GameObject someObject;
7    thePosition = transform.TransformPoint(2, 0, 0);
8
9    Transform cam = Camera.main.transform;
10   Vector3 cameraRelativeRight = cam.TransformDirection (Vector3.right);
11
12   Transform target;
13   bool enemyNear = false;
14
15   void Start()
16   {
17     Instantiate(someObject, thePosition, someObject.transform.rotation);
18   }
19
20   void Update()
21   {
22     transform.Rotate(Vector3.right * Time.deltaTime);
23
24     if(enemyNear == true)
25       transform.LookAt(target);
26     else
27       transform.LookAt(Vector3.zero);
28   }
29 }
```

Listing 4.18: TransformExample.cs

In Listing 4.18, `someObject` is created (l. 6) and instantiated in `Start()` to the right of the current object (l. 17). Then, a camera `cam` and `cameraRelativeRight` are created in order to calculate the x-axis relative to the camera (ll. 9-10). In `Update()` the object (player) is slowly rotated around its x-axis at one degree per second (l. 22). The variable `enemyNear` indicates if an enemy is near. If it is `true`, the player is looking at this enemy called `target` (ll. 24-25), otherwise she or he is looking at `Vector3.zero` which is a shorthand writing for `Vector3(0, 0, 0)`.

4.4.11 Quaternions

Quaternions are compact, do not suffer from gimbal lock and can easily be interpolated. Unity internally uses them to represent all rotations. Most often Quaternions take existing rotations (for example, from the `Transform`) and use them to construct new rotations (for example, to smoothly interpolate between two rotations). Some useful static methods are listed in Listing 4.19:

```
1  class Quaternion
2  {
3    ...
4    static Quaternion Inverse(Quaternion rotation);
5    static Quaternion Euler(float x, float y, float z);
6    static Quaternion Euler(Vector3 euler);
7    static Quaternion LookRotation(Vector3 forward, Vector3 upwards =
         Vector3.up);
8    static Quaternion Slerp(Quaternion from, Quaternion to, float t);
9    ...
10 }
```

Listing 4.19: Quaternion.cs

Scripting example:

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class QuaternionExample : MonoBehaviour
5  {
6    Quaternion rotation;
7    Transform target;
8    float distance;
9
10   void Update()
11   {
12     if(Input.GetKeyDown("i"))
13       transform.rotation = Quaternion.Inverse(target.rotation);
14     else if(Input.GetKeyDown("e"))
15       transform.rotation = Quaternion.Euler(0, 30, 0);
16     if(distance < 30.0f)
17       LookAtEnemy();
18   }
19
20   void LookAtEnemy()
21   {
22     rotation = Quaternion.LookRotation(target.position -
           transform.position);
23     transform.rotation = Quaternion.Slerp(transform.rotation, rotation,
           Time.deltaTime);
24   }
25 }
```

Listing 4.20: QuaternionExample.cs

The listed methods are implemented in Listing 4.20. If the player presses 'I', `Inverse()` sets the transform of the player to have the opposite rotation of the target (for example an enemy) (ll. 12-13). `Euler()` returns a rotation that rotates x, y, z degrees around the x, y, z axis. So, when the player presses 'E' the rotation 30 degrees around the y-axis happens (ll. 14-15). When the player is near an enemy, the method `LookAtEnemy()` is called (ll. 18-19). In `LookAtEnemy()` (ll. 20-24) `LookRotation()` creates a rotation with the specified `forward` and `upwards` directions. In this example, the player then always looks at the enemy's position (`target`) (l. 24). `Slerp()` spherically interpolates between `from` and `to` by `t`, so it interpolates between the two rotations (l. 25).

Part III

Conception and Implementation of the Action Adventure Scout COD (Caravan of the Damned)

Chapter 5

Scout COD Requirement Definitions

Scout COD (Caravan of the Damned) is a linear 3D action adventure in which the player impersonates a Scout, who needs to manage and lead a group of survivors through a devastated world. With the aid of this group, the Scout is able to avoid the eradication of humanity. The world was mainly destroyed by the managarms - a big and dangerous type of insects, which were created by a unfortunate accident in a laboratory. The Scout and his group have to survive the nature, other survivors and of course the managarms.

The player can select several landmarks on a general map of the continent. Different instances, which are 3D worlds, are provided to the player inside those landmarks. The player can then enter a instance, in which the actual game finally takes place.

In the present prototype, the character controls of the Scout and the combat system inside an instance are realized. This chapter defines the requirement definitions of the two systems in order to design the complete game in a second step.

Figure 5.1 shows a concept art of the Scout and a managarm in order to get an insight of the mood of *Scout COD*.

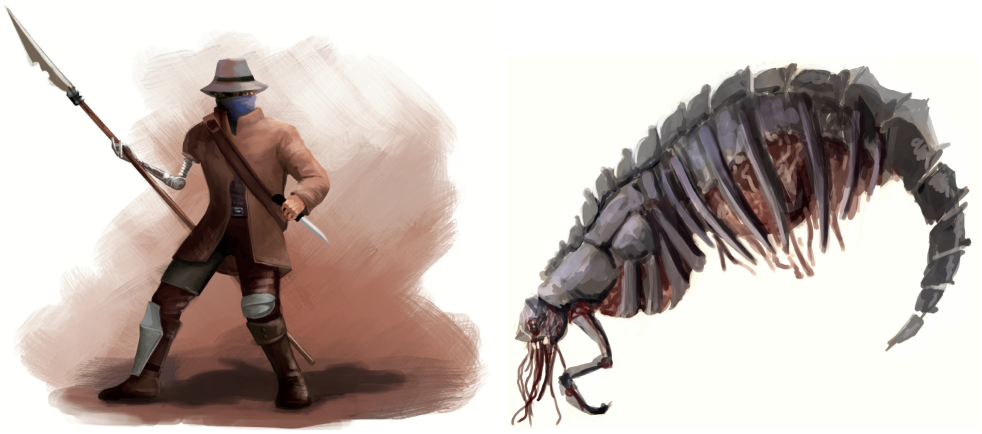


Figure 5.1: Concept art of the Scout and a managarm [1]

[1] Concept art by Jennifer Lehmann

5.1 Character Controls

The player can control the Scout by 'WASD' or arrow keys on the keyboard. Most gamers use 'WASD' because it is easier accessible as other keys. But there are also some people, such as lefties, who prefer using the arrow keys. In the following content only 'WASD' is used to describe character controls for a clear view but the arrow keys behave analogue.

In the following content, 'pressing' means that the player pressed the button only once, 'holding' means that he or she is constantly pressing the key.

The character moves forwards by pressing 'W', leftwards by pressing 'A', rightwards by pressing 'D', and backwards by pressing 'S'. Moving backwards means that the Scout turns around and walks towards the camera. The character moves diagonally forwards or backwards when some keys are combined. For example, when the player is holding 'W' and 'D' simultaneously, the character moves diagonally forwards and rightwards at the same time.

Additionally, the player can influence the movement of the Scout by using the mouse. The player character always is moving in the direction of the mouse cursor.

Idle

If there is no player input, the Scout is idling while the camera is freely movable. This gives the player the opportunity to view the player character from different perspectives.

After six seconds without any input, a random idle animation will be played. This motion is one out of three possible animations in the idle pool (looking around, yawning and rotating robot hand). After the random animation has finished, the standard idle animation starts again for six seconds, then another random idle animation will be played, and so on. If the player makes any input, the idle animations interrupt immediately and the camera moves behind the Scout.

The endurance of the Scout regenerates very quickly during the idling phases.

Jog

If the player holds 'WASD', he or she is jogging. Jogging is the standard locomotion state of the Scout.

The endurance regenerates slowly in this state.

Walk

It is possible to switch from jogging to walking by pressing 'Caps Lock' once and continues holding 'WASD'. If the key is pressed again, the Scout stops walking.

If the player jumps, walking will be cancelled and switched back to jogging as well as running bursts off walking. After the Scout finished running he will not walk but jog.

The walking Scout regenerates his endurance fast.

Run

By additionally holding 'Shift', the Scout is running. He stops running and continues jogging by realising the 'Shift' key.

The endurance value decreases in the meantime. If the endurance value reaches zero, the character stops and the player is unable to control the Scout anymore. This state is displayed by an exhaust animation. The Scout will be again controllable again when the exhaust animation has finished. The endurance yet has to be regenerated to the value of 40 in order to be able to run again. Up to this point, the player can at most jog.

Sneak

The Scout is also able to sneak by holding 'Ctrl' and 'WASD' at the same time. If 'Ctrl' is released, the player continues jogging.

By pressing the 'C' key, the character ducks. Then it is possible to control the character with 'WASD'. By pressing 'C', 'Ctrl' or 'Space', the character switches to jogging and by holding 'shift' directly to running.

Endurance regenerates equally to jogging while sneaking.

Jump

The player jumps by pressing the 'Space Bar'. He or she jumps in the particular direction if 'WASD' keys are used additionally.

The exhaust value varies depending on the current locomotion in which the player wants to jump. While idling the character just will do a little hopper. The jumping animations are equal for walk, jog and run but charges different exhaust values. The faster the Scout moves the more he will be exhausted.

Camera

The Action Adventure includes a third-person camera. The player should walk into the direction of the mouse cursor. If he or she does not move, the player is able to rotate the camera around the character in order to observe him.

5.2 Combat System

The combat system defines the facilities and the process of combat operations in the game.

Attacks

The Scout is competent to execute attacks without moving meanwhile.

Light attacks are activated by clicking the left mouse button. The player can make a light attack three times in a row. They differ in the animation and decrease the Scout's endurance by 7 points. The forth attack is a heavy attack which does not reduce endurance. The attack chain ends with the forth attack or when there is no more attack within five seconds. Then the chain starts from its beginning with the first light attack. If endurance is lower than 7 points, only the first light attack is possible to execute. Every light attack causes 10 damage points to an enemy, the heavy attack 20.

The player is also able to do a heavy attack by holding the left mouse button. This attack needs a recharge time of two seconds to be executed. Therefore, this attack is called boost attack. The endurance is reduced by 20 points per boost attack. If the Scout's endurance is below 20 points, he is not able to do such an attack again. A boost attack damages an enemy by 20 points.

Block

The Scout can switch into a block position as long as the right mouse button is hold. While blocking no other player input is possible. The character can not block with less than 5 endurance points.

Enemy Spawning

At the beginning of the game, some enemies are already spawned. From time to time new enemies are spawned, in order to keep danger alive.

Enemy Detection

Depending on his current locomotion state, enemies detect the Scout in a defined area. The faster he moves the earlier he is detected. This concept represents the hearing of the enemies. If someone is running, he or she is louder than someone who is just standing still. As soon as the enemy notices the player, he will move forwards to the Scout and stop moving in front of him. The enemy follows the player when he or she tries to walk away until the Scout will reach a specific distance.

Fight

The Scout can attack an enemy until he is dead. Every time the rival is hit, it does a specific animation to symbolize that damage was taken. If the enemy has no more life left, he will die and the object will be removed from the world after a defined time. The enemy does not have its own AI, he does not struggle.

5.3 Overview of Player Input

Figure 5.2 and Table 5.1 summarize the player input for simplification. The coloured keys are those which represent possible player inputs. Same coloured keys result in the same movement state. Keys with * are only pressed once, the others are need to be pressed continuously.

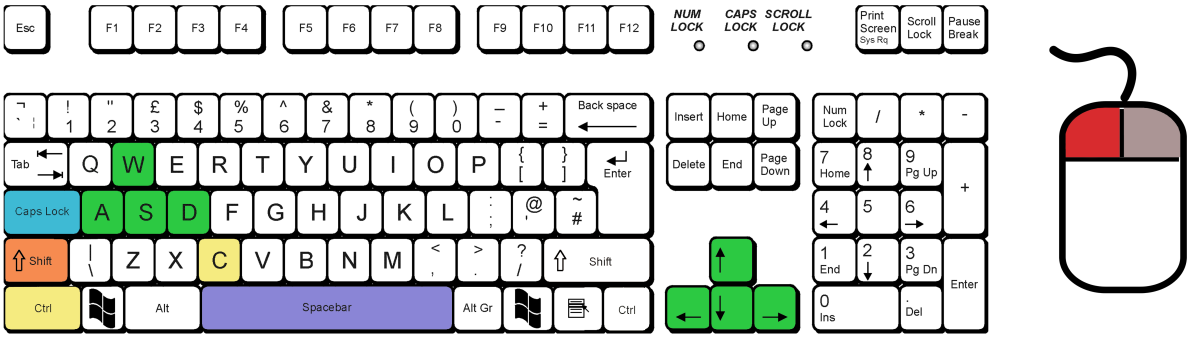


Figure 5.2: Keyboard and mouse input keys [2]

Keys	**Result**
WASD	Jog
Caps Lock* + WASD	Walk
Shift + WASD	Run
C* / Ctrl + WASD	Sneak
Space Bar*	Jump
Left Mouse Button*	Attack
Left Mouse Button	Boost attack
Right Mouse Button	Block

Table 5.1: Player input

[2] english Keyboard for Windows

Chapter 6

Scout COD Design

The Model View Controller (MVC) design pattern is used for the implementation of *Scout COD*. The three components of this pattern are described in this chapter.

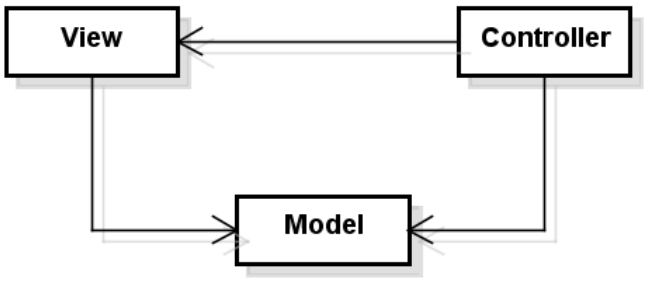


Figure 6.1: MVC pattern of *Scout COD*

Figure 6.3 depicts the MVC pattern. This variant is a small modification of the pattern by Gamma[1]. The original pattern can not be applied because of an automated refreshing of the view by the framework of Unity.

For example, the player looses 20 of his 100 life points because of an hit by an enemy. Foremost, the controller informs the model and the model updates the health value to 80. Then, the controller informs the view about the changed value so that the view knows that it has to execute an hit animation.

Summing up, the model represents all data of all game objects in the world. If there is any change from the controller by user input, the controller informs the model and the view. The view fetches the current data from the model.

[1] GAMMA et al. (1995).

6.1 Model

The model contains all data of every game object in the game. In reference to *Scout COD*, it consists of the following classes:

- `World` registers all players, enemies, subjects and NPCs[2]in a world. The Singleton pattern is applied in order to make sure that only one world can be instantiated.
- The class `Coor3D` represents the 3D coordinate of an object in the world.
- `WorldObject` is the root class of all game objects that are registerd in the world. It contains a name and the position.
- The child class `Subject` only exists for the sake of completeness, if subjects will be added to the game.
- The child class `NPC` only exists for the sake of completeness, if NPCs will be added to the game.
- The abstract class `LifeForm` contains the data of all characters. Every animated object disposes of health and endurance.
- The class `Player` inherits its data from its super class `LifeForm`. In addition, some variables for enemy detection and movement states are required.

The model also contains some useful enumeration types:

- `Enemy` inherits all data from `LifeForm` and does not command additional attributes.
- `Attack` enumerates all possible attacks that a character can execute.
- `Exhaust` enumerates the different exhaust values for the respective activities.
- `Movement` enumerates 3D movement values for all available locomotion states.
- `Regeneration` enumerates the endurance regeneration values depending on the current locomotion state.
- `Velocity` enumerates all available locomotion states.
- `ZoneFactor` enumerates the percentage values of every locomotion state in terms of enemy detection.

[2]A non-player character (NPC) in a game is any character that is not controlled by a player, for example an ally who have a dialogue with the player.

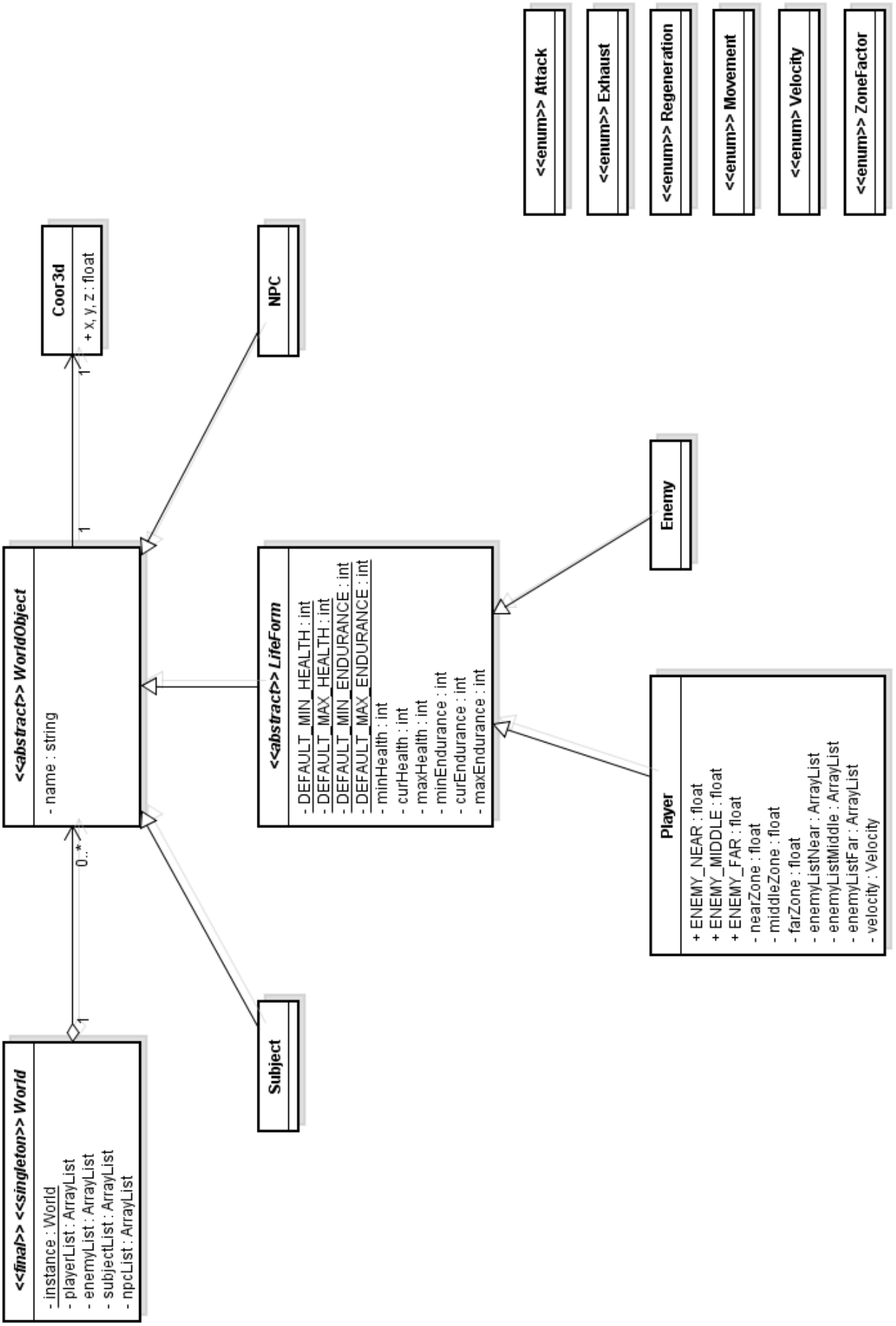


Figure 6.2: Model of *Scout COD*

6.2 View

The view represents the visual presentation in form of character animation. The head-up display (HUD) also belongs to the view. Unfortunately, Unity limits the proper implementation due to its requirements so that the HUD is part of the controller.

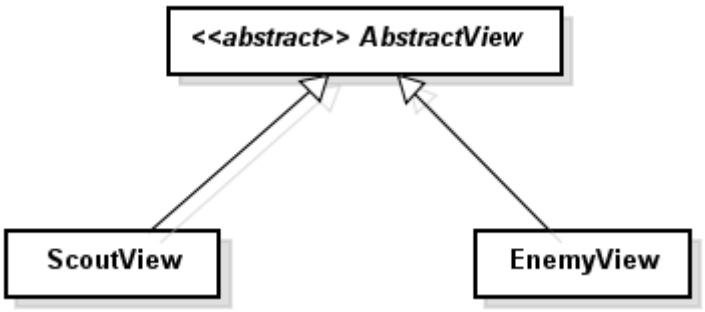


Figure 6.3: View of *Scout COD*

- `AbstractView` is a capsule around the animator component of Unity. The derived classes apply the facade pattern in order to facilitate an easier interface.
- `ScoutView` inherits the animator component by the super class `AbstractView`. The class administrates the animations of the player character. The animations are controlled by player input. So if there is any input the view updates its current animation.
- `EnemyView` updates the animations of an enemy. These animations are not controlled by the player directly but he or she interacts with the enemy, the player also controls the enemy behaviour indirectly.

6.3 Controller

The controller is triggered every frame when there is any user input. All classes that are part of the controller are inherited by Unity's base class `MonoBehaviour`.

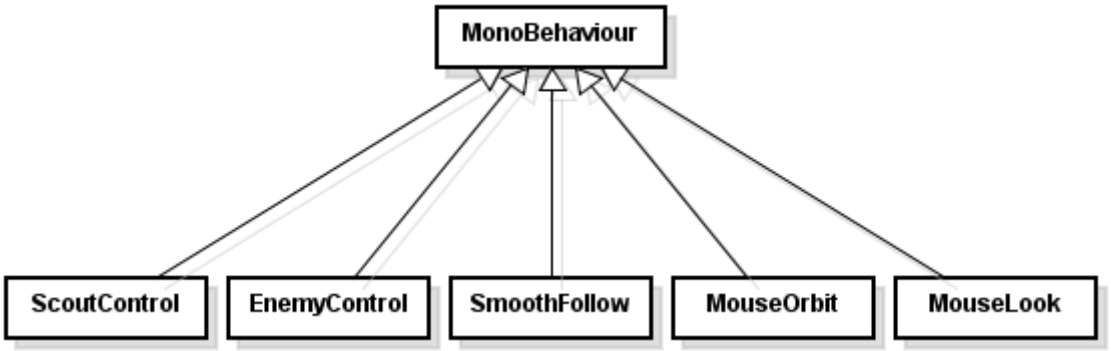


Figure 6.4: Controller of *Scout COD*

- `ScoutControl` is responsible for all player input. This class includes the implementation of different locomotion opportunities of the Scout, interactivity with an enemy, fighting, camera controlling and the HUD. The class works mainly together with `Player` and `ScoutView`.

- The class `EnemyControl` deals with the behaviour of enemies. The class works mainly together with `Enemy` and `EnemyView`.
- `SmoothFollow` allows a third-person view. That means that the camera is following the player wherever he or she is going.
- With `MouseLook` the player is able to look and then move into the direction of the mouse cursor.
- `MouseOrbit` rotates the camera around the player character.

Auxiliary, two Unity scripts, written in JavaScript, are used in the implementation of the needed physics referring to character movement:

- `CharacterMotor` provides a basis for various different types of character control. This highly complex script is the engine that drives a third person character. Especially the class `CharacterMotorMovement` helps to make a character moving, falling or halting.
- `FPSInputController` allows the player to move backward and forward using the vertical axis keys and to move side-to-side with the horizontal axis keys. The script works in tandem with `CharacterMotor` and provides input information to it.

Chapter 7

Scout COD Implementation

The following chapter deals with the prototype implementation the Action Adventure *Scout COD*.

First of all, the Unity Scene and its GameObjects are considered. Afterwards, all Components that are attached to the Scout Controller are shortly described.

The animations of the game characters are managed inside State Machines. The State Machine of the Scout is explained step by step. States, parameters and transitions are illustrated in state diagrams and the functionalities are described via code snippets. In addition to the locomotion states, the Scout disposes of endurance which needs to computed in scripts. The camera has also different functionalities with are shortly described at the end of this section.

The motions of attacking and blocking are also part of the Scout's complex State Machine. Enemy animations are also set up in another State Machine. Enemies detect a player within a certain area. These areas are explained too. If an enemy is attendant, he walks towards the player and then the Scout can start a combat. The enemy will react but not do anything. If this enemy is destroyed, new ones can be spawned in order to keep danger alive.

7.1 Unity Set

7.1.1 Scene

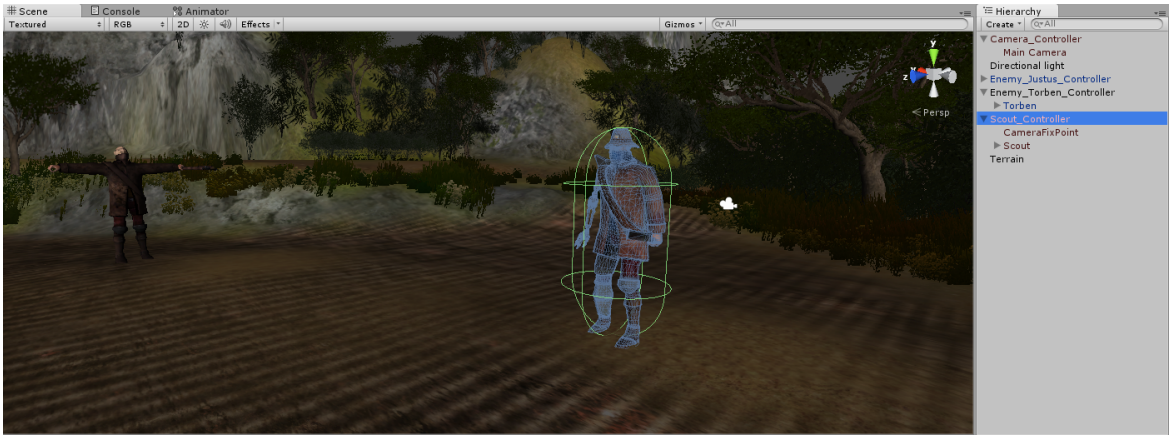


Figure 7.1: Unity scene

Figure 7.1 shows a scene extract of *Scout COD* in Unity. All GameObjects, that are part of the game, are listed in the Hierarchy:

- The *Camera_Controller* is the parent of the *Main Camera*. All scripts that affect camera control issues are attached to this object. The camera is detectable behind the Scout and symbolized with a white camera icon.
- *Directional light* brings light into the darkness.
- *Enemy_Justus_Controller* contains the mesh of enemy Justus. Justus is a modified mesh of the Scout.
- *Enemy_Torben_Controller* behaves analogous to Justus.
- The player character Scout is placed inside the *Scout_Controller*.
- The so-called *tower instance*[1]is used in this Unity project. The Terrain only includes the landscape but not elements such as buildings.

7.1.2 Import

All characters and animations of the Scout and the enemies were created in the 3D software package Maya[2]. The files were exported to FBX in order to import them into Unity. Unity creates an animation including the Avatar and the motion in addition to character joints.

Both *looping* and *non-looping* animations are used. In the settings, all animations have to be baked into pose. If not, some bugs, concerning rotation and position, can appear. In case of locomotion animations (idle, walk, jog, run, sneak) *Loop Time* and *Loop Pose* need to be checked so that the character repeats the motion until there is no more player input. Otherwise, animations that will be played only once (jump, exhaust, attack) do not make use of this functionality.

[1]level design by Dominik Kirner
[2]character animation by Max Schumayer

7.2 Character Controls

7.2.1 Scout Components

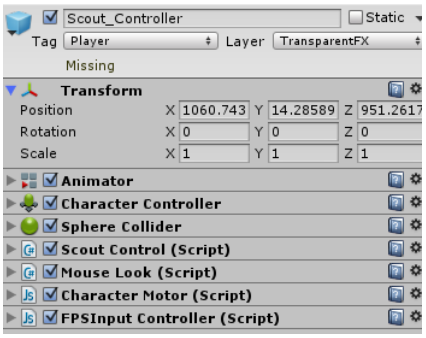


Figure 7.2: Transform

Figure 7.2 shows all components that are attached to the player character including Mecanim, Colliders and scripts. Like every GameObject, the Scout disposes of the `Transform` component that represent the current position, rotation and scale. All child objects move relative to this Controller. The following figures look closely at each component. The *Scout_Controller* receives the tag *Player* which is a reference in scripts independent of changing the name of the object.

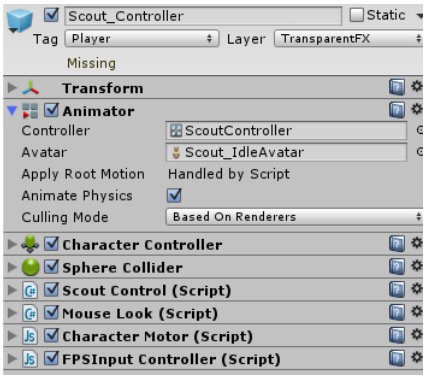


Figure 7.3: Animator

The *Scout_Controller* needs an Animator component (see Figure 7.3) in order to apply Unity's powerful Mecanim system. The created Animator Controller and the Avatar that was created during importing are attached to this component. By a double click on the Controller the Animator Windows opens and states, parameters and transitions can be added. *Apply Root Motion* allows controlling animations via scripts. The checked property *Animate Physics* tells the objects to interact with physics, without it, bugs like falling through terrain Colliders, can occur.

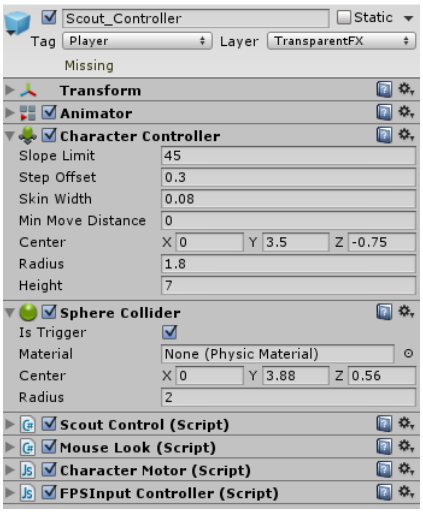


Figure 7.4: Character Controller

The Scout gets a Character Controller component (see Figure 7.1 and Figure 7.4) because Rigidbodies are not physically realistic for third-person player control. The Controller carries out the movement but is constrained by collisions. These properties are implemented in the scripts `CharacterMotor` and `FPSInputController` by Unity.

The Sphere Collider acts as trigger for the combat system (more information in Section 7.3).

7.2.2 Input Manager

Foremost, in Unity's Input Manager all axes and keys that are utilized for player input are named (see Table 7.1). Scripts can reference to these button names in order to take advantage of reusability. The horizontal and vertical axis are defined because of standard input via 'WASD' or arrow keys. The positive button represent the positive axis, the negative button the negative axis. 'Shift', 'Ctrl' and 'Space' also receive button names that represent their functionality.

Name	**Key(s)**
Horizontal	W, S
Vertical	D, A
Run	Left Shift
Sneak	Left Ctrl
Jump	Space

Table 7.1: Button and axis denotation

7.2.3 State Machine

Figure 7.5 features the State Machine of the player character.

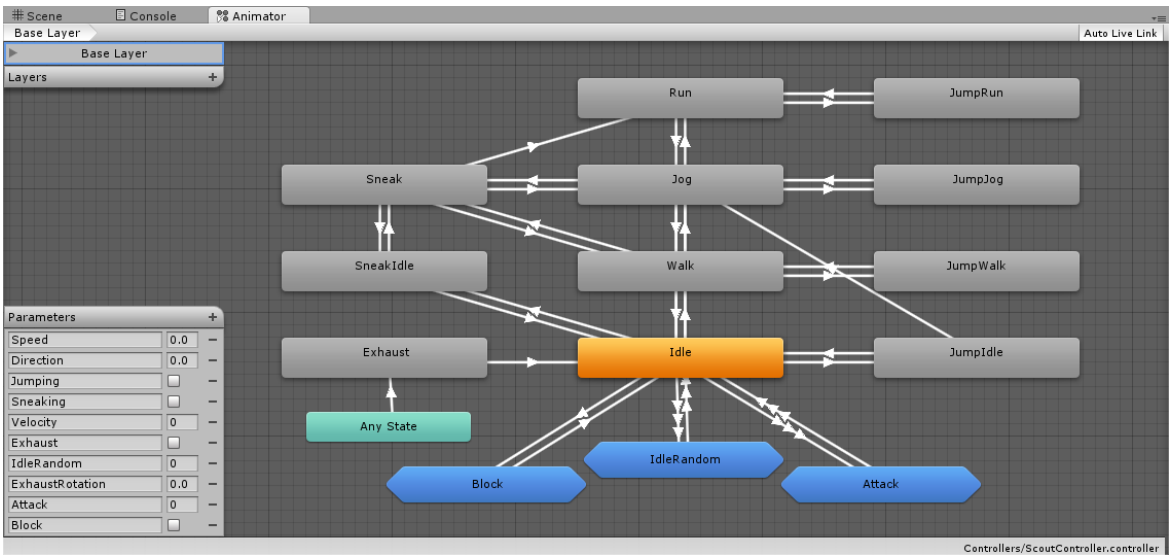


Figure 7.5: Animator extract of locomotion states

The first two parameters inside the Animator Controller are `Speed` and `Direction` which represent the input axes. `Speed` represents the vertical axis, `Direction` the horizontal one. They are both `float` parameters in order to make use of them inside a Blend Tree. All locomotion states contain Blend Trees.

The movement were originally relocated inside a 1D Blend Tree. The parameter `Direction` controlled player input for 'A' and 'D'. But when two keys, for example 'W' and 'D', were hold at the same time, the Scout just moved to the right. But he should walk diagonally rightwards. Therefore, this blend type is not helpful in this matter. The `Speed` parameter is also necessary for combining two keys.

The numerous motions are listed inside a 2D Freeform Directional Blend Tree (see Figure 7.6). `Speed` and `Direction` serve as blending parameters. Input axes have three valid values: 1 (positive keys), 0 (no input) and -1 (negative keys). Table 7.2 and 7.3 give an overview of the associated keys.

For instance, *Scout_Jog_Left* have the values -1 and 0. That means that this motion is played when the player presses only 'A'. If he or she presses 'W' and 'D' simultaneously, both parameters get an value of 1 and the animation *Scout_Jog_Right_Sideways* is played. The blue tetragons give a better imagination of in which direction the player is moving to. *Scout_Jog* would practically have the values 0 and 1, but it has 0 and 0 because Unity uses the forward motion as default motion. The Blend Trees for *Walk*, *Run* and *Sneak* include analogue animations and values.

Direction	key
1	D
0	-
-1	A

Table 7.2: Direction values

Speed	key
1	W
0	-
-1	S

Table 7.3: Speed values

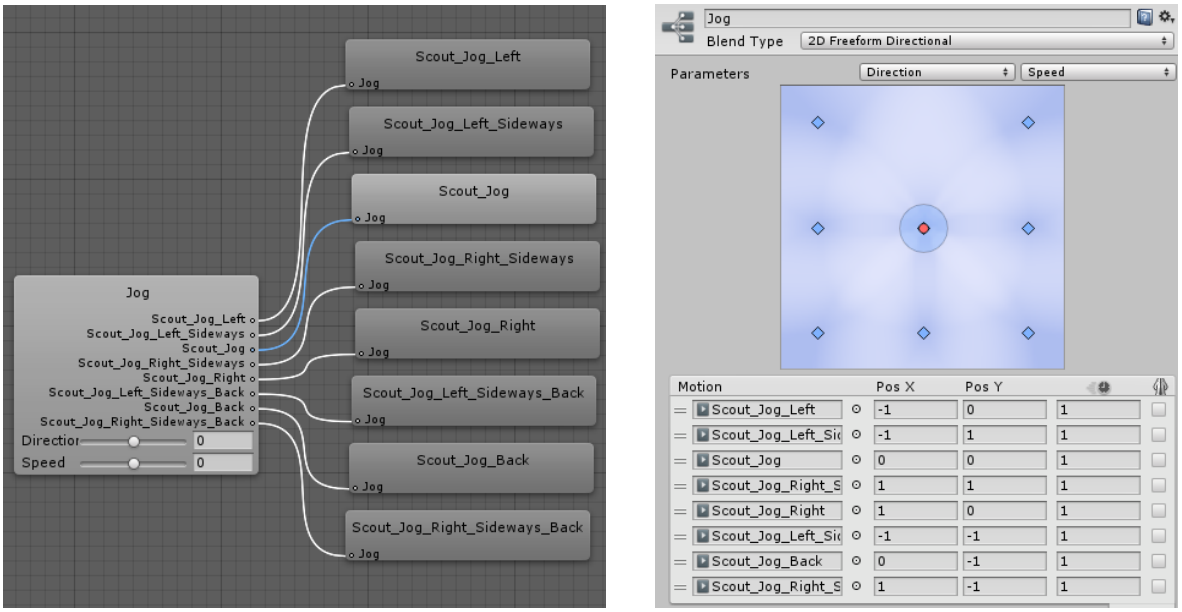


Figure 7.6: Blend Tree of locomotion state 'Jog'

It would be also conceivable to rotate the character by 45 degrees without these big Blend Trees but scripting does not allow the smooth transitions between two animations. Walking to the left and to the right in the next frame would look totally unnatural. Therefore, Blend Trees are qualified for different motions inside the same locomotion state.

`Velocity` is an `integer` parameter that gives every locomotion state an own value. Therewith, it is possible to set up transitions between states.

The `bool` parameter `Sneaking` indicates if the corresponding key is pressed or not. Because the character can duck with 'Ctrl' or 'C' without moving, this state needs this additional parameter. `Jumping` is a boolean parameter that states if the character is jumping or not.

Animator Access

Scripts access parameters via the Animator component. The class `ScoutView` contains the properties in which all parameters are defined per getter and setter methods (see Listing 7.1).

```
1  public class ScoutView : AbstractView
2  {
3    ...
4    public float Speed
5    {
6      get { return animator.GetFloat("Speed"); }
7      set { animator.SetFloat("Speed", value); }
8    }
9
10   public float Direction
11   {
12     get { return animator.GetFloat("Direction"); }
13     set { animator.SetFloat("Direction", value); }
14   }
15
16   public int Velocity
17   {
18     get { return animator.GetInteger("Velocity"); }
19     set { animator.SetInteger("Velocity", value); }
20   }
21
22   public bool Sneaking
23   {
24     get { return animator.GetBool("Sneaking"); }
25     set { animator.SetBool("Sneaking", value); }
26   }
27
28   public bool Jumping
29   {
30     get { return animator.GetBool("Jumping"); }
31     set { animator.SetBool("Jumping", value); }
32   }
33
34   public int IdleRandom
35   {
36     get { return animator.GetInteger("IdleRandom"); }
37     set { animator.SetInteger("IdleRandom", value); }
38   }
39
40   public Vector3 MoveTo(Transform t)
41   {
42     Vector3 newPosition = t.position;
43     newPosition.z += Speed * Time.deltaTime;
44     newPosition.x += Direction * Time.deltaTime;
45     return t.position = newPosition;
46   }
47   ...
48 }
```

Listing 7.1: Locomotion properties (ScoutView.cs)

The view permits the class `ScoutControl` to access these properties. Next to the fact that the view contains these methods because of visual presentation, it is convenient to have the Animator changes outside of controller scripts. If there are any changes inside the State Machine, the parameter properties just need to be changed once in the view.

The parameters for 'WASD' input `Speed` and `Direction` (ll. 4-14) can be set with the method `SetFloat()`. If there is any specified player input, transitions update parameter values. `GetFloat()` returns the particular value. The parameters `Velocity`, `Sneaking`, `Jumping` and `IdleRandom` make use of the methods `Get/SetInteger()` (ll. 16-20) or `Get/SetBool()` (ll. 22-26) that are part of the Animator component.

`MoveTo()` is called immediately when the player character is moving from one point to another (ll. 28-34). This method computes the new position of the Scout every time in seconds it took to complete the last frame.

```
1  public class ScoutControl : MonoBehaviour
2  {
3    private Animator anim;
4    private ScoutView scoutView;
5    private Player scout;
6    ...
7    void Start ()
8    {
9      anim = GetComponent<Animator>();
10     ...
11     scout = new Player("Chefscout",
           Util.UnityToModelCoor(transform.position));
12     scoutView = new ScoutView(anim);
13     ...
14   }
15
16   void OnAnimatorMove()
17   {
18       scout.Position =
             Util.UnityToModelCoor(scoutView.MoveTo(transform));
19   }
20 }
```

Listing 7.2: Animator access (ScoutControl.cs)

Listing 7.2 shows the declared variables of `ScoutControl` that are used in nearly all methods that are explained in this chapter. `anim` (l. 3) is a variable that allows access to the Animator Component. With the help of `scoutView` (l. 4), the controller can update the view and `scout` (l. 5) is an object of its base class `Player`.

In the `Start()` message the Animator component is loaded (l. 9). Then, a new player `scout` is created (l. 11). The constructor (see l. 11 and Listing 7.3) gives a player a name and a position. Finally, `scoutView` is created by the constructor of `ScoutView` (see l. 12 and Listing 7.4).

The Unity message `OnAnimatorMove()` (ll. 16-19) is called at each frame after the State Machine and the animations have been evaluated and computes the position of the Scout using the `MoveTo()` method (l. 18) of `ScoutView`.

```
1  public class Player : LifeForm
2  {
3    ...
4    public Player(string name, Coor3D pos) : this()
5    {
6      Name = name;
7      Position = pos;
8    }
9    ...
10 }
```

Listing 7.3: Player constructor (Player.cs)

```
1  public class ScoutView : AbstractView
2  {
3    ...
4    public ScoutView (Animator animator) : base(animator) {}
5    ...
6  }
```

Listing 7.4: ScoutView constructor (ScoutView.cs)

Idle, Walk, Jog, Run and Sneak

Listing 7.5 shows the implemented player input in terms of locomotion. Unity's message `FixedUpdate()` allows a constantly player movement.

```
1  public class ScoutControl : MonoBehaviour
2  {
3    ...
4    private static bool capslockPressed = false;
5    private static bool cPressed = false;
6    ...
7    void FixedUpdate ()
8    {
9      float h = Input.GetAxis("Horizontal");
10     float v = Input.GetAxis("Vertical");
11     Velocity vel = Velocity.Idle;
12
13     scoutView.Direction = h;
14     scoutView.Speed = v;
15
16     if(Input.GetKeyDown(KeyCode.CapsLock))
17       capslockPressed = !capslockPressed;
18
19     if(v != 0 || h != 0)
20     {
21       vel = Velocity.Jog;
22
23       if(Input.GetButton("Run"))
24       {
25         vel = Velocity.Run;
26         capslockPressed = false;
27       }
28       if (cPressed)
29         vel = Velocity.Sneak;
```

```
30          if(capslockPressed)
31            vel = Velocity.Walk;
32
33        vel = Sneaking(v, h, vel);
34        scoutView.Velocity = (int)vel;
35        scout.Velocity = vel;
36        ...
37        }
38    }
39    ...
40 }
```

Listing 7.5: FixedUpdate() (ScoutControl.cs)

First of all, some variables are declared. The bool variable `capslockPressed` (l. 4) is used for *Walk* in order to get to know if 'Capslock' has been pressed or not. It is set to `false` at the beginning. The same concept is applied for `cPressed` referring sneaking. In `FixedUpdate()` the two input axes get `float` variables (`h`, `v`) for the purpose of using them in another context. `vel` is the variable that describes the current locomotion state. It is set to `Idle` by default (l. 10). The different values, which the enumeration type `Velocity` can accept, are listed in Listing 7.6. `Direction` and `Speed` need to be updated in the view (ll. 12-13).

```
1 public enum Velocity
2 {
3   Idle = 0, Sneak = 1, Walk = 2, Jog = 3, Run = 4
4 }
```

Listing 7.6: Velocity.cs

If the player presses 'Capslock', `capslockPressed` immediately gets the opposite value that it had before (ll. 15-16). If `capslockPressed` was true, then it is false and vice versa.

The player is jogging as soon as he or she is holding either 'W', 'A', 'S' or 'D' (ll. 19-21). If the player additionally holds the Run button ('Shift'), he is running (ll. 23-27). Once the player is running, `capslockPressed` is set to `false` so that the Scout never walks again after running. If the player is pressing 'C', `vel` gets the value for sneaking (ll. 28-29). The same holds true for `capslockPressed` based on walking (ll. 30-31).

Then, the method `Sneaking()` is called separately in order to reduce code inside of `FixedUpdate()`. Therefore, the parameters `v`, `h` and `vel` have to be transferred (l. 33). `vel` needs to be updated both in the view and the controller (ll. 34-35).

Listing 7.7 now presents the called method `Sneaking()`. The player can sneak either by holding the Sneak button ('Ctrl') or pressing 'C'.

If the player presses 'C', `cPressed` immediately gets the opposite value that it had before (l. 8). If `cPressed` was true, then it is false and vice versa. So, if the player presses 'c' he ducks (*SneakIdle*) (l. 10). The Scout is sneaking as soon as 'WASD' are added. If the player is ducking and presses 'Ctrl', the player character stands up and goes back to *Idle* (ll. 21-28).

The player is also able to sneak with 'Ctrl'. He or she ducks at the moment this button is hold down (l. 16). In addition with 'WASD' the player can move and sneak (ll. 18-19).

```
1  public class ScoutControl : MonoBehaviour
2  {
3    ...
4    Velocity Sneaking(float v, float h, Velocity vel)
5    {
6      if(Input.GetKeyDown("c"))
7      {
8        cPressed = !cPressed;
9
10       if(cPressed) scoutView.Sneaking = true;
11       else         scoutView.Sneaking = false;
12     }
13
14     if(Input.GetButton("Sneak"))
15     {
16       scoutView.Sneaking = true;
17
18       if(v != 0 || h != 0)
19         vel = Velocity.Sneak;
20
21       if (cPressed)
22       {
23         scoutView.Sneaking = false;
24         cPressed = !cPressed;
25       }
26     }
27     else if(!cPressed)
28       scoutView.Sneaking = false;
29
30     return vel;
31   }
32   ...
33 }
```

Listing 7.7: Sneak() (ScoutControl.cs)

Movement Speed

The Scout now can do all locomotion that differ in the respective animations. But whatever motion is playing, the Scout always moves equally through the 3D world. The `CharacterMotor` script of Unity computes the movement speed of a character. Because it haggles over a script written in JavaScript, getter and setter methods are added in order to change the variables in `ScoutControl` (see Listing 7.8).

```
1  ...
2  function SetForwardSpeed(x:float)    { movement.maxForwardSpeed = x;      }
3  function SetSidewaysSpeed(x:float)   { movement.maxSidewaysSpeed = x;     }
4  function SetBackwardsSpeed(x:float)  { movement.maxBackwardsSpeed = x;    }
5
6  function GetForwardSpeed() : float   { return movement.maxForwardSpeed;   }
7  function GetSidewaysSpeed() : float  { return movement.maxSidewaysSpeed;  }
8  function GetBackwardsSpeed() : float{ return movement.maxBackwardsSpeed;}
```

Listing 7.8: getter and setter methods for movement speed (CharacterMotor.js)

The particular speed values are listed in the enumeration list `Movement` (see Listing 7.9). These values make locomotion even more realistic.

```
1 public enum Movement
2 {
3   Idle = 0, Sneak = 3, Walk = 5, Jog = 10, Run = 15
4 }
```

Listing 7.9: Movement.cs

Listing 7.10 shows the method `SetCharacterMovement()` that is called inside of `FixedUpdate()`. As a parameter the method needs the velocity because the computation is dependant of it. For every locomotion another movement speed is realized (ll. 22-41).

Afterwards, the current values are changed by setter methods (ll. 43-45).

```
1  public class ScoutControl : MonoBehaviour
2  {
3    ...
4    private CharacterMotor motor;
5    ...
6    void Start ()
7    {
8      ...
9      motor = GetComponent<CharacterMotor>();
10     ...
11   }
12
13   void FixedUpdate()
14   {
15     ...
16     SetCharacterMotorMovement(vel);
17     ...
18   }
19
20   void SetCharacterMotorMovement(Velocity vel)
21   {
22     Movement motorSpeed = Movement.Idle;
23
24     switch(vel)
25     {
26     case Velocity.Idle :
27       motorSpeed = Movement.Idle;
28       break;
29     case Velocity.Sneak :
30       motorSpeed = Movement.Sneak;
31       break;
32     case Velocity.Walk :
33       motorSpeed = Movement.Walk;
34       break;
35     case Velocity.Jog :
36       motorSpeed = Movement.Jog;
37       break;
38     case Velocity.Run :
39       motorSpeed = Movement.Run;
40       break;
41     }
```

```
42
43      motor.SetForwardSpeed((float)motorSpeed);
44      motor.SetSidewaysSpeed((float)motorSpeed);
45      motor.SetBackwardsSpeed((float)motorSpeed);
46    }
47  }
```

Listing 7.10: SetCharacterMotorMovement() (ScoutControl.cs)

Jumping

The method `Jumping()` is also called in `FixedUpdate()` (l. 7). If the player presses the Jump button 'Space', the Animator parameter `Jumping` turns to `true` (ll. 20-25). Jumping while sneaking is not possible. If the player presses 'space' meanwhile, the Scout cancels sneaking and goes back to *Idle* (ll. 15-19).

```
1  public class ScoutControl : MonoBehaviour
2  {
3    ...
4    void FixedUpdate()
5    {
6      ...
7      Jumping();
8      ...
9    }
10
11   void Jumping()
12   {
13     if(Input.GetButtonDown("Jump"))
14     {
15       if (cPressed)
16       {
17         scoutView.Sneaking = false;
18         cPressed = !cPressed;
19       }
20       else
21       {
22         ...
23         scoutView.Jumping = true;
24         ...
25       }
26     }
27     else
28     {
29       scoutView.Jumping = false;
30     }
31   }
32   ...
33 }
```

Listing 7.11: Jumping() (ScoutControl.cs)

State diagram

Figure 7.7 summarizes all transitions between the different states referring locomotion.

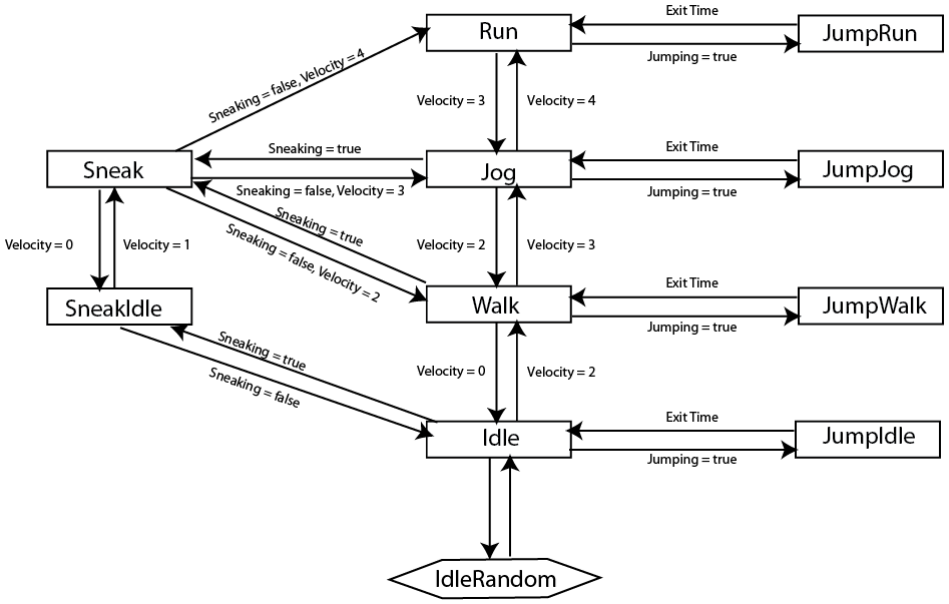


Figure 7.7: Locomotion state diagram

Idle Pool

The idle pool of the Scout consists of three different animations (four with the standard idle). They are grouped inside the Sub-State Machine `IdleRandom` (see Figure 7.8) and controlled by the `integer` parameter `IdleRandom`. Every motion has its own value (*Idle* = 0, *RotatingHand* = 1, *Yawning* = 2 and *LookingAround* = 3). The transistions from the *Base Layer* to the idle animations use these values to know which motion is addressed.

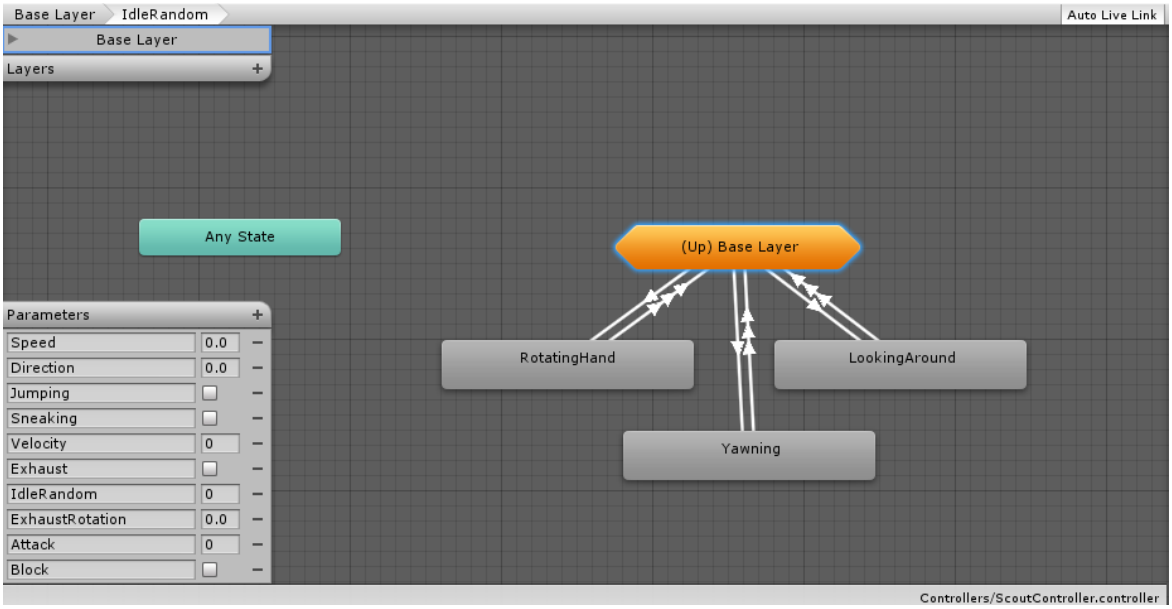


Figure 7.8: Sub-State Machine IdleRandom

Every state needs two transitions back to the *Base Layer*. The first one ends the animation when it has been finished playing (*Exit Time*). The second transition cancel a random animation if the player wants to move via 'WASD' (`Velocity` is greater than zero).

Figure 7.9 illustrates all transitions between the different states referring locomotion.

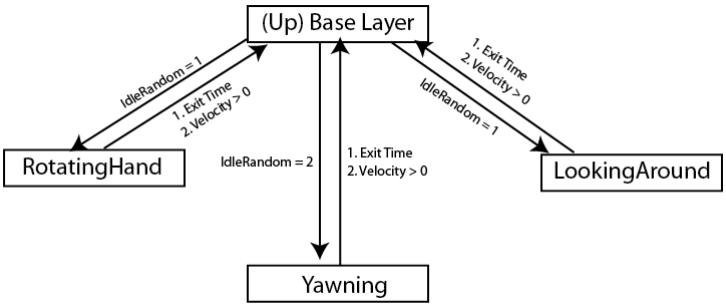


Figure 7.9: State diagram of Sub-State Machine IdleRandom

Listing 7.12 implements this idle pool. Foremost, five variables and constants are declared. Constants are practical in case of any changes. `INTERVAL_IDLE` indicates after which time in seconds an idle animation will be played, in this case every six seconds (l. 4). `MIN_RANDOM` and `MAX_RANDOM` are the minimum and maximum values of an idle animation that is set by `IdleRandom` inside the State Machine (ll. 5-6). The boolean `fromIdleStandard` tells Unity if the current animation is the standard idle animation (`true`) or one of the random ones (`false`) (l. 7). `inIdlePool` states if the Scout is executing one of the random animations (is in idle pool) or not.

The coroutine `IdlePool()` plays random idle animations. First of all, the variable `inIdlePool` gets `true` inside this method (l. 30). The `yield` statement suspends the coroutine execution for six seconds (l. 39). If the current motion is the standard idle animation, the next one will be one of the three random idle animations. `Random.Range()` returns a random animation by its `IdleRandom` parameter. Afterwards, the view is updated (ll. 43-44). The Scout is executing the standard idle animation after the random one has played for six seconds (ll. 46-49). Then, the standard idle motion plays again for six seconds, then another random idle is played and so on (example: *StandardIdle* - *Yawning* - *StandardIdle* - *LookingAround* - *StandardIdle* - *Yawning* - *StandardIdle* - *RotatingHand*). If the player does any input (`Input.anyKey`), all is set back to default (ll. 32-36).

`FixedUpdate()` calls the method `ControlIdlePool()` (l. 13). Without this method, there would be a problem with the interval time of six seconds. For instance, the character is idling and then he moves for four seconds. The idle pool would already start after the two left seconds but not after the desired six seconds. It would be just a lucky coincidence if the Scout would wait really six seconds for executing a random motion. Therefore, Coroutines are used in order to control the idle pool. If the Scout is performing his random idle animations and then the player does any input, the Coroutine `IdlePool()` is stopped (ll. 19-26). If the player now stops, `IdlePool()` will be restarted.

```
1  public class ScoutControl : MonoBehaviour
2  {
3    ...
4    private const int INTERVAL_IDLE = 6;
5    private const int MIN_RANDOM = 1;
6    private const int MAX_RANDOM = 3;
7    private bool fromIdleStandard = true;
8    private bool inIdlePool = false;
9    ...
10   void FixedUpdate ()
11   {
12     ...
13     ControlIdlePool();
14     ...
15   }
16
17   void ControlIdlePool()
18   {
19     if (inIdlePool && Input.anyKey || scout.IsAlarm)
20     {
21       StopCoroutine("IdlePool");
22       inIdlePool = false;
23     }
24     if (!inIdlePool)
25       StartCoroutine("IdlePool");
26   }
27
28   IEnumerator IdlePool()
29   {
30     inIdlePool = true;
31
32     if(Input.anyKey)
33     {
34       scoutView.IdleRandom = 0;
35       fromIdleStandard = true;
36     }
37     else
38     {
39       yield return new WaitForSeconds(INTERVAL_IDLE);
40
41       if(fromIdleStandard)
42       {
43         int state = Random.Range(MIN_RANDOM, MAX_RANDOM + 1);
44         scoutView.IdleRandom = state;
45       }
46       else
47       {
48         scoutView.IdleRandom = 0;
49       }
50
51       fromIdleStandard = !fromIdleStandard;
52     }
53     inIdlePool = false;
54   }
55   ...
56 }
```

Listing 7.12: ControlIdlePool() and IdlePool() (ScoutControl.cs)

7.2.4 Endurance

Computation

The Scout disposes of the player attribute endurance. Activities like running or jumping reduce his endurance while others regenerate it. The combat system also makes use of endurance. The respective states and their values both for exhaust and regeneration are listed in Listing 7.25 and 7.14.

```
1 public enum Exhaust
2 {
3   Run = 10, End = 25, JumpDefault = 10, JumpIdle = 3, JumpWalk = 5,
        JumpJog = 10, JumpRun = 15, JumpSneak = 0, ...
4 }
```

Listing 7.13: Exhaust.cs

```
1 public enum Regeneration
2 {
3   Idle = 10, Walk = 7, Jog = 5, Sneak = 5
4 }
```

Listing 7.14: Regeneration.cs

In Listing 7.15, `ComputeEndurance()` is called every two seconds since the game will has been started (ll. 6-11). This is possible because of `Invoke()` inside of the `Start()` message (l. 9). The method is already implemented in the `Player` model class to which the controller accesses via the `player` variable.

```
1  public class ScoutControl : MonoBehaviour
2  {
3    ...
4    private const int INTERVAL_COMPUTE_ENDURANCE = 2;
5    ...
6    void Start ()
7    {
8      ...
9      InvokeRepeating("ComputeEndurance", 0, INTERVAL_COMPUTE_ENDURANCE);
10     ...
11   }
12
13   void ComputeEndurance()
14   {
15     scout.ComputeEndurance();
16   }
17   ...
18 }
```

Listing 7.15: ComputeEndurance() (ScoutControl.cs)

Endurance is computed for every locomotion state (see Listing 7.16, ll. 8-13). In this method, only running reduces the Scout's endurance. But the player can regenerate endurance as well. Idling regenerates endurance most rapidly, then walking and finally sneaking and jogging. The exhaust and regeneration values match with those of the mentioned enumeration types.

```
1  public class Player : LifeForm
2  {
3    ...
4    public void ComputeEndurance()
5    {
6      switch(velocity)
7      {
8        case Velocity.Idle  : Endurance += Regeneration.Idle; break;
9        case Velocity.Sneak : Endurance += Regeneration.Jog;  break;
10       case Velocity.Walk  : Endurance += Regeneration.Walk; break;
11       case Velocity.Jog   : Endurance += Regeneration.Jog;  break;
12       case Velocity.Run   : Endurance -= (int)Exhaust.Run;  break;
13       default: break;
14     }
15   }
16   ...
17 }
```

Listing 7.16: ComputeEndurance() (Player.cs)

Jumps

Jumping also costs endurance depending on the particular locomotion. In Listing 7.17, the class `ScoutView` creates variables for every locomotion state which act like an identity (ll. 4-8). The name of the perspective state is converted into an `integer`. `CurrentState` returns these states by means of the name hashes (ll. 10-13).

```
1  public class ScoutView : AbstractView
2  {
3    ...
4    public static int idleState = Animator.StringToHash("Base Layer.Idle");
5    public static int walkState = Animator.StringToHash("Base Layer.Walk");
6    public static int jogState = Animator.StringToHash("Base Layer.Jog");
7    public static int runState = Animator.StringToHash("Base Layer.Run");
8    public static int sneakState = Animator.StringToHash("Base
       Layer.Sneak");
9    ...
10   public int CurrentState
11   {
12     get { return animator.GetCurrentAnimatorStateInfo(0).nameHash; }
13   }
14   ...
15 }
```

Listing 7.17: Hash tags for states (ScoutView.cs)

The method `Jumping()` adds these hash tags in order to compute variant exhaust values (see Listing 7.18). Inside the `if`-statements, the current state is retrieved and a value of the enumeration type `Exhaust` is matched (ll. 15-27). Jumping while idling does not cost much endurance, but the faster the player moves the quicker he or she will be exhausted. So, persistent jumps while running are not advisable because the Scout will loose his endurance very fast.

With the help of hash tags, it is also possible to implement the following: If the player is walking and then jumps, walking will be cancelled and he continues jogging (l. 26).

```
1  public class ScoutControl : MonoBehaviour
2  {
3    ...
4    void Jumping()
5    {
6      if(Input.GetButtonDown("Jump"))
7      {
8        ...
9        Exhaust exhaustValue = Exhaust.JumpDefault;
10       int currentState = scoutView.CurrentState;
11
12       if (currentState == ScoutView.idleState)
13         exhaustValue = Exhaust.JumpIdle;
14       else if (currentState == ScoutView.jogState)
15         exhaustValue = Exhaust.JumpJog;
16       else if (currentState == ScoutView.runState)
17         exhaustValue = Exhaust.JumpRun;
18       else if (currentState == ScoutView.sneakState)
19         exhaustValue = Exhaust.JumpSneak;
20       else if (currentState == ScoutView.walkState)
21       {
22         exhaustValue = Exhaust.JumpWalk;
23         capslockPressed = false;
24       }
25     }
26   }
27   ...
28 }
```

Listing 7.18: Jumping() (ScoutControl.cs)

Exhaust

```
1  public class ScoutView : AbstractView
2  {
3    ...
4    public static int exhaustState = Animator.StringToHash("Base
       Layer.Exhaust");
5    ...
6    public bool Exhaust
7    {
8      get { return animator.GetBool("Exhaust"); }
9      set { animator.SetBool("Exhaust", value); }
10   }
11
12   public float ExhaustRotation
13   {
14     get { return animator.GetFloat("ExhaustRotation"); }
15     set { animator.SetFloat("ExhaustRotation", value); }
16   }
17   ...
18 }
```

Listing 7.19: Exhaust (ScoutView.cs)

If the player has no more endurance, an exhaust animation is played. Listing 7.19 shows an extract of the `ScoutView` class in which the properties `Exhaust` and `ExhaustRotation` are defined (ll. 6-16). They are both parameters of the State Machine. In addition, the hash tag for the exhaust state is declared (l. 4).

Listing 7.20: If the current endurance attains a value of zero, the exhaust animation is played (l. 12). During this motion, there is no more player input allowed and possible (l. 11).

Figure 7.10 presents the transitions in reference to the exhaust state.

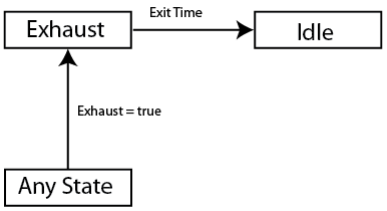


Figure 7.10: State diagram of Exhaust

Any State is not flawless in Unity 4. But it is possible to achieve the desired result with some detours. If the parameter `Exhaust` is set to `true` in the Animator, the exhaust animation loops all the time and never ends. Therefore, the parameter needs to be set back to `false` as soon as the animation starts (ll. 16-17). The Scout will now be back to idle after the motion has been finished.

```
1  public class ScoutControl : MonoBehaviour
2  {
3    ...
4    private bool exhausted = false;
5    ...
6    void FixedUpdate ()
7    {
8      ...
9      if(scout.Endurance == 0)
10     {
11       Input.ResetInputAxes();
12       scoutView.Exhaust = true;
13       exhausted = true;
14     }
15     ...
16     if (scoutView.CurrentState == ScoutView.exhaustState)
17       scoutView.Exhaust = false;
18   }
19   ...
20 }
```

Listing 7.20: No endurance (ScoutControl.cs)

The character always turns first forwards, before it starts to play the exhaust animation. If the player ran backwards before, it is strange that the Scout does not execute the motion also in the same direction. The reason for this is that there is no Blend Tree for Exhaust. But Blend Trees base on player input. With a trick it is possible to regulate this problem. A new `float` parameter `ExhaustRotation` is created as a means to an end.

Figure 7.11 depicts the Blend Tree and its settings in the Inspector. Because `ExhaustRotation` is the only blending parameter, a 1D Blending is sufficient. Eight exhaust animations relating to the perspective direction are listed. Every motion gets a Threshhold value between -135 and 180. These values represent the rotation of the character.

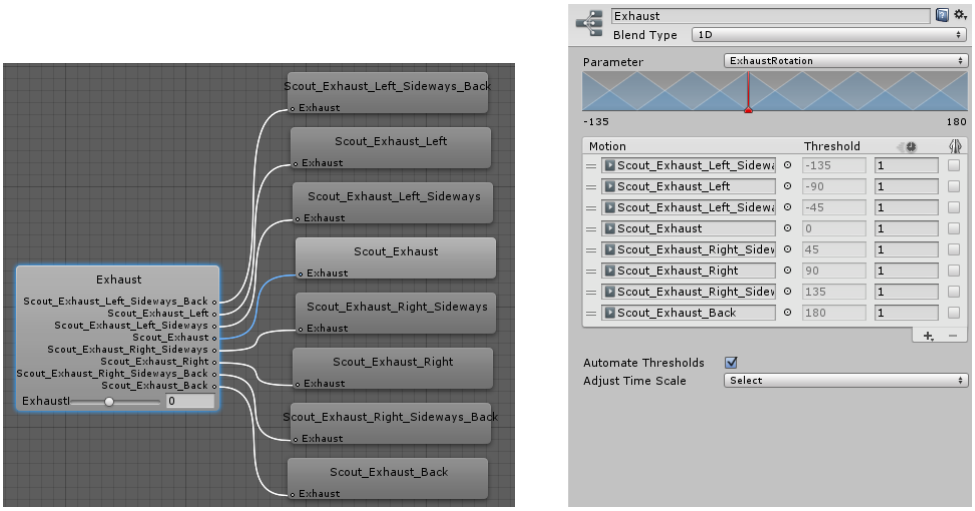


Figure 7.11: Blend Tree of Exhaust

Listing 7.21: The values are assigned to the particular player input inside the method `ExhaustRotation()` (ll. 11-21) that is called by `FixedUpdate()` (l. 7). The method needs the input axes as parameters. All key combinations allocate the perspective values. For instance, if `v` is 0 and `h` is 1 (only 'D' hold) the appropriate value results in 90. By means of the Blend Tree, Unity now knows that it has to play the *Scout_Exhaust_Right* motion.

```
1  public class ScoutControl : MonoBehaviour
2  {
3    ...
4    void FixedUpdate ()
5    {
6      ...
7      ExhaustRotation(v, h);
8      ...
9    }
10
11   void ExhaustRotation(float v, float h)
12   {
13         if(v == 0 && h == -1)  scoutView.ExhaustRotation =  -90f;
14     else if(v == 1 && h == -1)  scoutView.ExhaustRotation =  -45f;
15     else if(v == 1 && h == 0)   scoutView.ExhaustRotation =    0f;
16     else if(v == 1 && h == 1)   scoutView.ExhaustRotation =   45f;
17     else if(v == 0 && h == 1)   scoutView.ExhaustRotation =   90f;
18     else if(v == -1 && h == -1) scoutView.ExhaustRotation = -135f;
19     else if(v == -1 && h == 0)  scoutView.ExhaustRotation =  180f;
20     else if(v == -1 && h == 1)  scoutView.ExhaustRotation =  135f;
21   }
22   ...
23 }
```

Listing 7.21: ExhaustRotation() (ScoutControl.cs)

Head-Up-Display

Listing 7.22 guides the endurance bar that is placed in the HUD of the game. Graphical elements are created inside of the `OnGUI()` message of Unity that is automatically called for rendering and handling GUI events. Two variables of the `Texture2D` are defined. This data type includes all 2D graphics, in this case the endurance bar. Two several textures are required, one for current endurance the other for the background with no endurance (see Figure 7.12). They are set to `public` in order to assign the textures in the Inspector of the game engine.

```
1  public class ScoutControl : MonoBehaviour
2  {
3    ...
4    public Texture2D enduranceBar, curEnduranceBar;
5    ...
6    void OnGUI()
7    {
8      GUI.DrawTexture(new Rect(10, 50, scout.MaxEndurance * 5, 50),
           enduranceBar);
9      GUI.BeginGroup(new Rect(10, 50, scout.Endurance * 5, 50));
10     GUI.DrawTexture(new Rect(0, 0, scout.MaxEndurance * 5, 50),
           curEnduranceBar);
11     GUI.EndGroup();
12   }
13   ...
14 }
```

Listing 7.22: OnGUI() (ScoutControl.cs)

Figure 7.12: Textures for endurance bar

`OnGUI()` first positions the bar in the left upper corner (10 left, 50 top) and then draws the texture of the background fives times longer as `MaxEndurance` (by default 100) is and with the height of 50 (l. 9). `curEnduranceBar` is positioned equally.

This message achieves that the player can always see its state of current endurance (see Figure 7.13). In this example, the player has approximately 30 per cent of its endurance left.

Figure 7.13: Current endurance bar

7.2.5 Camera Control

Listing 7.23 illustrates the control of the three camera scripts that are attached to the *Camera_Controller*.

```
1  public class ScoutControl : MonoBehaviour
2  {
3    ...
4    private SmoothFollow smoothFollow;
5    private MouseLook mouseLookX, mouseLookY;
6    private MouseOrbit mouseOrbit;
7    public GameObject cameraObject, mainCamera;
8    ...
9    void Start ()
10   {
11     ...
12     mouseLookX = GetComponent<MouseLook>();
13     mouseOrbit = cameraObject.GetComponent<MouseOrbit>();
14     smoothFollow = cameraObject.GetComponent<SmoothFollow>();
15     mouseLookY = mainCamera.GetComponent<MouseLook>();
16     ...
17   }
18
19   void FixedUpdate ()
20   {
21     ...
22     CameraControl(vel);
23     ...
24   }
25
26   void CameraControl(Velocity vel)
27   {
28     if(vel == Velocity.Idle)
29     {
30       if(!scout.IsAlarm)
31       {
32         mouseOrbit.enabled = true;
33         smoothFollow.enabled = false;
34         mouseLookX.enabled = false;
35         mouseLookY.enabled = false;
36       }
37     }
38     else
39     {
40       mouseOrbit.enabled = false;
41       smoothFollow.enabled = true;
42       mouseLookX.enabled = true;
43       mouseLookY.enabled = true;
44     }
45   }
46   ...
47 }
```

Listing 7.23: CameraControl() (ScoutControl.cs)

The method `CameraControl()` which is called inside of `FixedUpdate()` specifies which scripts are enabled or disabled in diverse circumstances. Hence, it needs the parameter `vel`.

- The class `SmoothFollow` creates a third-person camera. It is located behind the player and follows him wherever he or she moves. The script is only enabled when the player is moving (l. 33, l. 41).
- `MouseLook` makes the player move in the direction of the mouse cursor. The script is attached to both the camera and the Scout GameObject. The horizontal mouse movement is attached to the player, the vertical to the camera. This allows the player both to look up and down and move to the left or right depending on the position of the mouse cursor. `MouseLook` is disabled if the player is situated in idle state and is not in danger. In a fight the script is again enabled in order to rotate in the direction of the enemy (ll. 34-35, ll. 42-43).
- By means of `MouseOrbit` the player can rotate the camera around the character. He or she can now admire the idle animations in other perspectives. The script is only enabled when the Scout is idling and not in the proximity of an enemy (l. 32, l. 40).

7.3 Combat System

7.3.1 Attacks

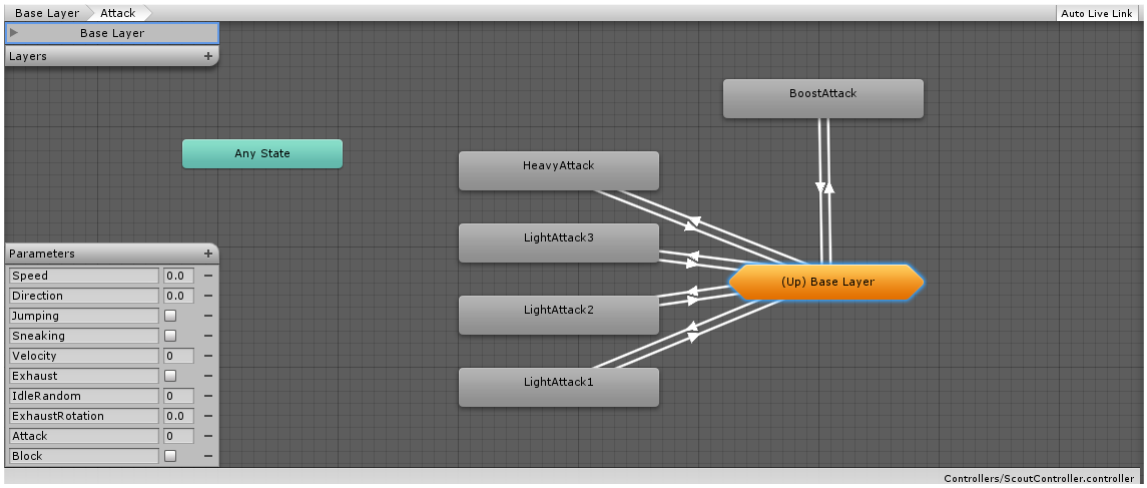


Figure 7.14: Sub-State Machine Attack

Figure 7.15 illustrates Sub-State Machine `Attack`. The parameter `Attack` specifies the different attack possibilities of the Scout, which is used in the transitions from *Base Layer* to the attack states. The current transition ends when the motion is over. The appropriate state diagram is depicted in Figure 7.15. Attacking is also possible without moving because otherwise the animations would contradict themselves. In order to attack while moving, additionally motions, which animate an attack and a locomotion together, ought to be created.

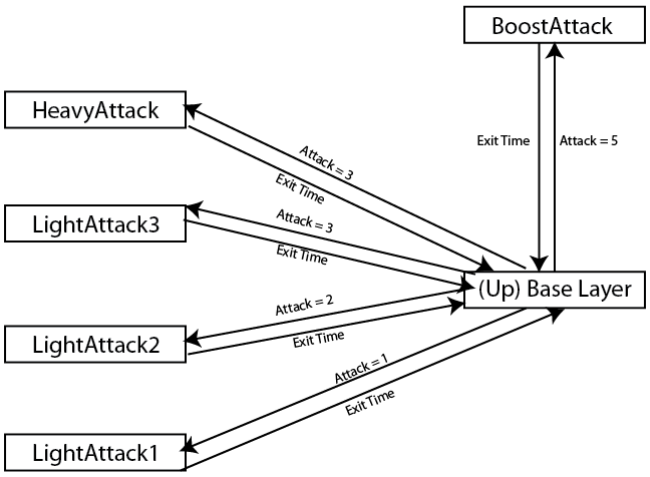


Figure 7.15: State diagram of Attacks

LightAttack 1, 2, 3 and *HeavyAttack* are part of an attack chain. They are played only if the previous motion has been executed. *BoostAttack* is handled separately.

The `integer` parameter `Attack` has to be added inside of the `ScoutView` (see Listing 7.24).

```
1  public class ScoutView : AbstractView
2  {
3    ...
4    public int Attack
5    {
6      get { return animator.GetInteger("Attack"); }
7      set { animator.SetInteger("Attack", value); }
8    }
9    ...
10 }
```

Listing 7.24: Attack property (ScoutView.cs)

Attacking reduces players endurance. The proper exhaust values are listed in the enumeration type `Exhaust` (see Listing 7.25).

```
1  public enum Exhaust
2  {
3    ... LightAttack = 7, HeavyAttack = 0, BoostAttack = 20, AttackMin =  7
4  }
```

Listing 7.25: Exhaust.cs

The Scout can execute overall five different attack animations by clicking the left mouse button. The `Attack` values are enumerated in Listing 7.26.

```
1  public enum Attack
2  {
3    None = 0, Light1 = 1, Light2 = 2, Light3 = 3, Heavy = 4, HeavyBoost = 5
4  }
```

Listing 7.26: Attack.cs

Listing 7.27 implements the method `AttackEnemy()` that is called by `FixedUpdate()` (l. 18). First of all, some variables need to be declared. `N_CHAIN_ATTACK` signifies the number of the animations in the attack chain, which is in this case four (l. 4). `INTERVAL_RESET_CUR_ATTACK` represents the time for the end of the attack chain (l. 5). The current attack is set to `None` (l. 6). This property originates from the enumeration type `Attack`. The `bool` variable `mouseClick` states if the mouse button has been clicked or not (l. 7).

All light attacks costs equally endurance (l. 20). The forth heavy attack does not reduce the endurance (ll. 32-33). The attack chain is updated when the current attack value is greater than 4. The chain restarts with *LightAttack1* again (ll. 22-23).

The method `ResetCurrentAttack()` sets the current attack back to *LightAttack1*. If there is no more player input for four seconds, the attack chain is cancelled and it starts with the first attack possibility and not with the third, for example (l. 5, l.12, ll. 43-45).

```
1  public class ScoutControl : MonoBehaviour
2  {
3    ...
4    private const int N_CHAIN_ATTACK = 4;
5    private const int INTERVAL_RESET_CUR_ATTACK = 4;
6    private Attack curAttack = Attack.None;
7    private bool mouseClick = false;
8    ...
9      void Start ()
10     {
11       ...
12       InvokeRepeating("ResetCurAttack", 0, INTERVAL_RESET_CUR_ATTACK);
13       ...
14     }
15   void FixedUpdate ()
16   {
17     ...
18     AttackEnemy();
19     ...
20   }
21
22   void AttackEnemy()
23   {
24     ...
25     if(mouseClick && scout.CanAttack(Exhaust.AttackMin))
26     {
27       Exhaust exhaust = Exhaust.LightAttack;
28       ...
29       if((int)++curAttack > N_CHAIN_ATTACK)
30         curAttack = Attack.Light1;
31
32       if (curAttack == Attack.Heavy)
33         exhaust = Exhaust.HeavyAttack;
34
35       ...
36       return;
37     }
38     else
39     {
40       scoutView.Attack = 0;
41     }
42   }
43   void ResetCurAttack()
44   {
45     curAttack = Attack.None;
46   }
47   ...
48 }
```

Listing 7.27: Attack() (ScoutControl.cs)

Listing 7.28 presents another opportunity to attack an enemy. Unity's `Update()` message calls the method `BoostAttack()` (l. 12). The code snippet shows that if the player is holding the left mouse button for two seconds (ll. 27-28), the boost attack will be executed (16-18). As contrasted with *HeavyAttack*, which is the same motion, *BoostAttack* reduces endurance.

```
1  public class ScoutControl : MonoBehaviour
2  {
3    ...
4    private const int ATTACK_BOOST_TIME = 2;
5    private float boostDownTime, boostHoldTime;
6    private bool boostHoldState = false;
7    private bool boost = false;
8    ...
9    void Update()
10   {
11     ...
12     BoostAttack();
13     ...
14   }
15
16   void BoostAttack()
17   {
18     ...
19     else if(Input.GetMouseButton(0) && boostHoldState)
20     {
21       boostHoldTime = Time.realtimeSinceStartup - boostDownTime;
22
23       if(boostHoldTime >= ATTACK_BOOST_TIME)
24         boost = true;
25     }
26     ...
27   }
28   ...
29 }
```

Listing 7.28: BoostAttack() (ScoutControl.cs)

7.3.2 Block

The `bool` parameter `Block` tells Unity if the player is currently blocking or not. This parameter needs to be set up in the `ScoutView` (see Listing 7.29).

```
1  public class ScoutView : AbstractView
2  {
3    ...
4    public bool Block
5    {
6      get { return animator.GetBool("Block"); }
7      set { animator.SetBool("Block", value); }
8    }
9    ...
10 }
```

Listing 7.29: Block property (ScoutView.cs)

The Scout has also the ability to block attacks by enemies. The Sub-State Machine `Block` contains three states with three different animations (see Figure 7.16). The first one, *Block-Start*, is a motion in which the Scout goes in position. *BlockHold* is a looping animation of the block state that is replayed until the mouse button is released. If the button finally is released, the last motion *BlockEnd* is played. The Scout breaks away from its block position.

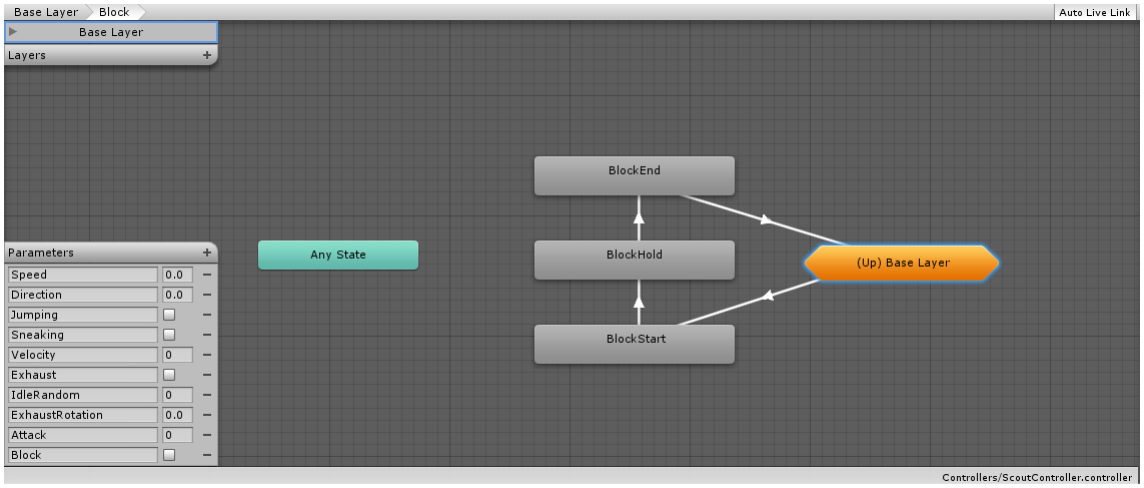


Figure 7.16: Sub-State Machine Block

Figure 7.17 summarizes the transitions in a state diagram.

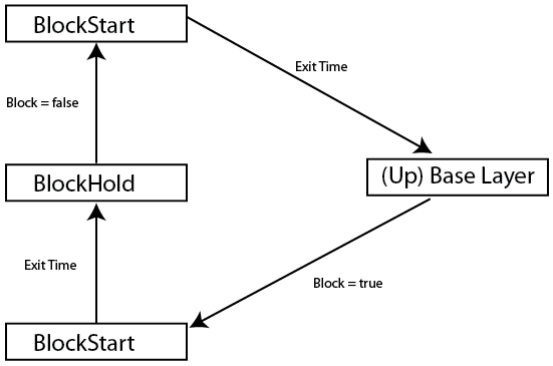


Figure 7.17: State diagram of Block

Listing 7.30 presents the method `Block()` that is called by `FixedUpdate()` (l. 7). It simply sets the `Block` property to `true` as long as the right mouse button is hold (l. 13).

```
1  public class ScoutControl : MonoBehaviour
2  {
3     ...
4     void FixedUpdate ()
5     {
6        ...
7        Block();
8        ...
9     }
10
11    void Block()
12    {
13       if(Input.GetMouseButton(1)) scoutView.Block = true;
14       else                        scoutView.Block = false;
15    }
16    ...
17 }
```

Listing 7.30: Block() (ScoutControl.cs)

7.3.3 Enemy State Machine

The enemy GameObject has four Components attached: `Transform`, Animator Component, Character Controller and the script `EnemyControl`. Enemies make also use of Unity's Mecanim. Figure 7.18 illustrates the State Machine of an enemy object.

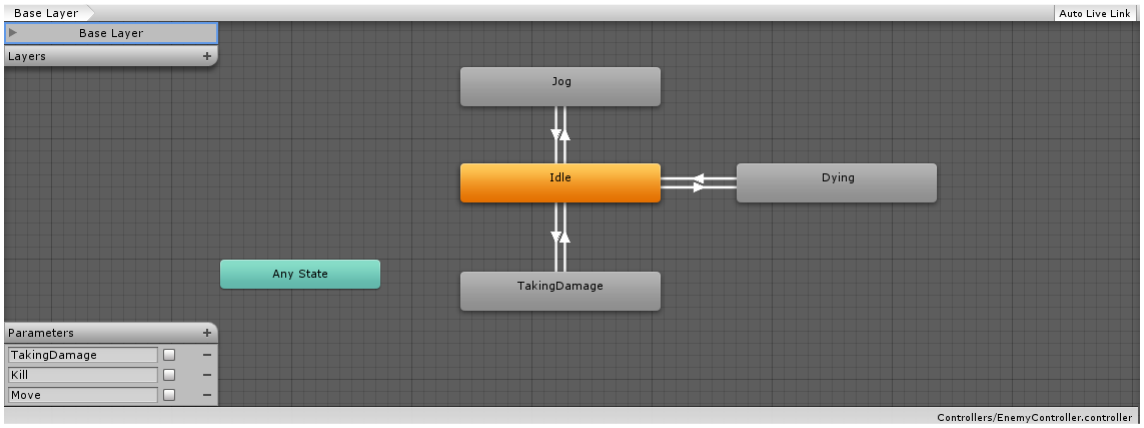


Figure 7.18: Enemy State Machine

Jog is equivalent to the movement of the Scout. *TakingDamage* is a motion that is played when an enemy is hit by the player. If the enemy is finally dead, a *Dying* 'motion', that was created with Rag Doll, is played. Three `bool` parameters states if the current state is active or not (see Figure 7.19).

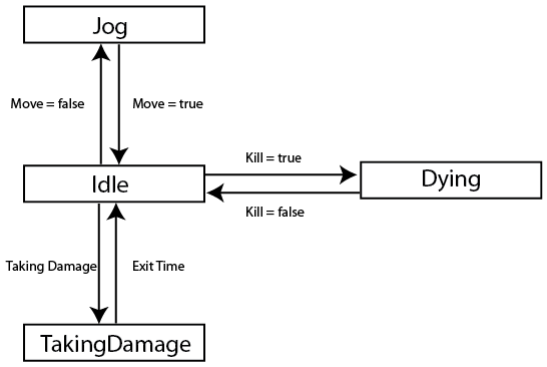


Figure 7.19: State diagram enemy

7.3.4 Detection Zones

Enemies are idling when there is no player in the proximity. If the player oversteps a specific distance, an enemy gets attentive. The detection zones are divided into three areas: near zone, middle zone and far zone. For instance, the game should not render enemies that are in the far zone. Considering the combat system, the near zone is important (see Listing 7.31).

```
1  public class Player : LifeForm
2  {
3    ...
4    public const float ENEMY_NEAR   = 30.0f;
5    private float nearZone;
6    private ArrayList enemyListNear;
7
8    public void CheckNearEnemies()
9    {
10     ArrayList newList = new ArrayList();
11
12     foreach(Enemy enemy in enemyListNear)
13     {
14       float distance = Coor3D.Distance(this.Position, enemy.Position);
15
16       if (distance > nearZone)
17       {
18         if (distance <= middleZone) enemyListMiddle.Add(enemy);
19         else if (distance <= farZone) enemyListFar.Add(enemy);
20         else {}
21       }
22       else
23         newList.Add(enemy);
24     }
25     enemyListNear.Clear();
26     enemyListNear = newList;
27   }
28
29   public bool IsAlarm
30   {
31     get { return enemyListNear.Count > 0; }
32   }
33
34   public float NearZone
35   {
36     get { return nearZone; }
37   }
38   ...
39 }
```

Listing 7.31: CheckNearEnemies() (Player.cs)

The variable `ENEMY_NEAR` specifies the distance of the near zone (l. 4). All enemies in the proximity are registered in a defined list (ll. 12-24). So `CheckNearEnemies()` register new enemies in the zone or unregister them if they leave it.

If there is more than one enemy inside of the near zone, the Scout is alarmed (ll. 29-32). Another property returns the `float` value of the near zone (ll. 34-37).

```
1  public enum ZoneFactor
2  {
3    Idle = 100, Sneak = 110, Walk = 120, Jog = 150, Run = 200
4  }
```

Listing 7.32: ZoneFactor.cs

The hearing and seeing faculties of humans and animals are also considered. The faster the player moves the earlier an enemy can see or hear him. The distance needs to be increased.

The factors depending on the current locomotion state are enumerated in Listing 7.32.

Listing 7.33 computes the values of `ZoneFactor`. The range of near is depending on the velocity of the Scout (l. 4). Foremost, the values are converted into percentage (ll. 10-27). The `SetData()` method then multiplies the new factors with the distance of the near zone (ll. 29-31).

```
1  public class Player : LifeForm
2  {
3    ...
4    private Velocity velocity = Velocity.Idle;
5    ...
6    private void SetData()
7    {
8      float f = 0.0f;
9
10     switch(velocity)
11     {
12     case Velocity.Idle:
13       f = (float) ZoneFactor.Idle / 100.0f;
14       break;
15     case Velocity.Sneak:
16       f = (float) ZoneFactor.Sneak / 100.0f;
17       break;
18     case Velocity.Walk:
19       f = (float) ZoneFactor.Walk/ 100.0f;
20       break;
21     case Velocity.Jog:
22       f = (float) ZoneFactor.Jog / 100.0f;
23       break;
24     case Velocity.Run:
25       f = (float) ZoneFactor.Run / 100.0f;
26       break;
27     }
28
29     nearZone = f * ENEMY_NEAR;
30     middleZone = f * ENEMY_MIDDLE;
31     farZone = f * ENEMY_FAR;
32     ...
33   }
34   ...
35 }
```

Listing 7.33: SetData() (Player.cs)

If the enemy can see or hear the Scout, he will walk towards him. The states *Idle* and *Move* in the enemies State Machine are connected with the `bool` parameter `Move` (see Listing 7.34.

```
1  public class EnemyView : AbstractView {
2
3    ...
4    public void Move()
5    {
6      animator.SetBool("Move", true);
7    }
8
9    public void StopMove()
10   {
11     animator.SetBool("Move", false);
12   }
13   ...
14 }
```

Listing 7.34: Move() and StopMove() (EnemyView.cs)

`MoveToPlayer()` in Listing 7.35 makes the enemies walking forward to the player and follow him. First of all, the scout is needed (l. 6) because an enemy needs him as a GameObject to move to. `distance` states the distance between an enemy and the Scout (l. 7). If the player now enters the near zone (l. 9), first the rotation of an enemy is computed (ll. 11-12). By means, the enemy know always rotate to the direction of the player. In a next step, the enemy will walk forwards orienting on the computed rotation. The enemy starts to walk forwards with a specific move speed (l. 14, ll. 24-28). The rotation and position are computed simultaneously.

If the enemy is close to the player he stops to walk (ll. 19-23). The `moveSpeed` is set back to zero and the animation will be stopped. Otherwise, they will not stop colliding. If the player then walks away, the enemy will follow as long this takes place inside of the near zone. If the Scout runs away, the enemy will loose its interest.

```
1  public class EnemyControl : MonoBehaviour
2  {
3    ...
4    public void MoveToPlayer()
5    {
6      Player scout = (Player) World.Instance.PlayerList[0];
7      float distance = Coor3D.Distance(enemy.Position, scout.Position);
8
9      if(distance < scout.NearZone)
10     {
11       Quaternion rotation = Quaternion.LookRotation(player.position -
             transform.position);
12       transform.rotation = Quaternion.Slerp(transform.rotation,
             rotation, Time.deltaTime * DAMPING);
13
14       Vector3 delta = transform.forward * enemy.MoveSpeed *
             Time.deltaTime;
15       transform.position += delta;
16
17       enemy.Position = Util.UnityToModelCoor(transform.position);
18
19       if(distance < Player.ENEMY_STOP)
20       {
21         enemy.MoveSpeed = 0.0f;
```

```
22            enemyView.StopMove();
23          }
24          else
25          {
26            enemy.MoveSpeed = 6.0f;
27            enemyView.Move();
28          }
29        }
30        else
31        {
32          if(enemy.MoveSpeed > 0f)
33          {
34            enemy.MoveSpeed = 0.0f;
35            enemyView.StopMove();
36          }
37        }
38        ...
39    }
40    ...
41 }
```

Listing 7.35: MoveToPlayer() (EnemyControl.cs)

7.3.5 Combat

As the State Machine has already shown, an enemy can take damage of the Scout and finally die. The parameters are set in Listing 7.36.

```
1  public class EnemyView : AbstractView {
2    ...
3    public void TakingDamageTrigger()
4    {
5      animator.SetTrigger("TakingDamage");
6    }
7
8    public void Dying()
9    {
10     animator.SetBool("Kill", true);
11     animator.enabled = false;
12   }
13   ...
14 }
15 }
```

Listing 7.36: TakingDamageTrigger() and Dying() (EnemyView.cs)

The Scout has additionally a Sphere Collider around his 'attack area' (see Figure 7.20). This Trigger enables hits when the enemy is in that zone.

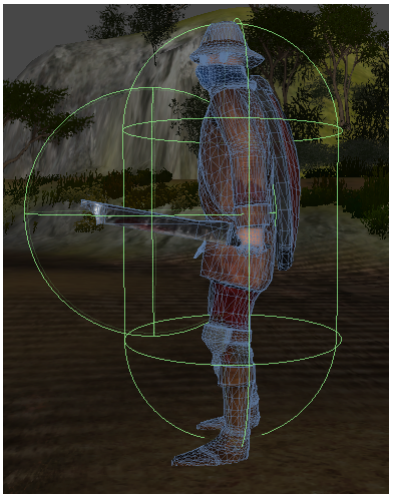


Figure 7.20: Attack Trigger Zone

```
1  public class EnemyControl : MonoBehaviour
2  {
3    ...
4    private const float REACTION_DELAY = 0.5f;
5    private const int DESTROY_ENEMY = 30;
6    ...
7    public void Hit(int damage, ScoutControl scout)
8    {
9      enemy.Hit(damage);
10
11     if (!enemy.IsDead)
12     {
13       StartCoroutine("InternalHit");
14     }
15     else
16     {
17       enemyView.Dying();
18       Invoke("DestroyEnemy", DESTROY_ENEMY);
19       World.Instance.Remove(enemy);
20     }
21   }
22
23   private IEnumerator InternalHit()
24   {
25     yield return new WaitForSeconds(REACTION_DELAY);
26     enemyView.TakingDamageTrigger();
27   }
28   ...
29 }
```

Listing 7.37: Hit() (EnemyControl.cs)

The `Hit()` method needs `damage` and the `scout` as its parameters in order to interact with `ScoutControl`. When the enemy is in the Trigger Zone of the Scout and the player executes an attack, the enemy gets a hit (l. 9). If the enemy is still alive after the hit, it executes the `TakingDamage` motion. But the animation should not start directly when the player is clicking a mouse button because this would look unnatural. The Coroutine `InternalHit` is called that waits half a second, before the enemy State Machine starts to play the damage

animation (l. 13, ll. 23-27).

The *Dying* animation is not really a animation. Dying is created by means of Rag Doll physics. This allows to assign several Colliders as bones to the skeleton of an enemy. The 'bones' make use of Rigidbodies physics. Rag Doll computes and creates a randomly dying animation on a physically level. If the enemy has no more life, it executes this 'motion' (l. 17). After 30 seconds the dead object will be removed out of the world (ll. 18-19).

7.3.6 Enemy Spawning

At the beginning of the game, some enemies are already spawned. After certain time intervals new enemies will be spawned. In Listing 7.38, `DoEnemySpawn()` is called in the `Start()` message via invoke. The two constants (ll. 4-5) define the start time and intervals. The first enemy will spawn in 90 seconds, then every 30 seconds a new enemy will spawn (l. 10).

Enemies spawn randomly in a particular area (ll. 17-18). If a clone of Torben or of Justus will be spawned is randomly defined too (l. 20). The spawn types are assigned via Prefabs in the Inspector. Finally, a new enemy is placed in the world (ll. 22-30).

```
1  public class ScoutControl : MonoBehaviour
2  {
3    ...
4    private const float START_SPAWN = 90.0f;
5    private const float INTERVAL_SPAWN = 30.0;
6    ...
7    void Start ()
8    {
9      ...
10     InvokeRepeating("DoSpawnEnemy", START_SPAWN, INTERVAL_SPAWN);
11     ...
12   }
13
14   void DoSpawnEnemy()
15   {
16
17     float dx = Random.Range(30.0f, 50.0f);
18     float dz = Random.Range(30.0f, 50.0f);
19
20     int spawnType = Random.Range(1, 3);
21
22     switch (spawnType)
23     {
24     case 1:
25       Instantiate(EnemyJustusSpawner, transform.position + new
           Vector3(dx, 0, dz) , Quaternion.identity);
26       break;
27     case 2:
28       Instantiate(EnemyTorbenSpawner, transform.position + new
           Vector3(dx, 0, dz) , Quaternion.identity);
29       break;
30   }
31   ...
32 }
```

Listing 7.38: DoEnemySpawn() (ScoutControl.cs)

Part IV

Conclusion

Chapter 8

Conclusion

8.1 Summary

Game development is a relatively new and rapid growing industry. Even more and more people want to join the games branch and start to develop their own games. But a lot of them relinquish after a short time.

The purpose of the present thesis was to develop a prototype for the 3D Action Adventure *Scout COD*. The implementation includes both character controls and combat system.

Unity 3D was chosen as game engine. Mecanim, Unity's uniquely powerful and flexible animation system brings characters to life with incredibly natural and fluid motion. Unity even offers much more features and is very popular in the games industry. Unity also provides a huge community. All in all, the engine is a good choice to learn game development.

The implementation of the prototype was organized in three phases:

- First of all, requirement definitions needed to be discussed before someone could have even started programming.
- By reference to those requirement definitions architecture of the single components and their relations to each other can be designed. Design and architecture patterns cater for a better concept.
- Finally, the prototype of the game was implemented.
  - Character controls include all states a player character can execute via player input. This includes standard locomotion like walk and run or random idle animations. The states reduces or regenerate the endurance of the player. Camera controls are also part of character controls because the player can control the camera with the mouse.
  - Combat system contains enemies that walk towards the player or the player even kills an enemy. Enemies can be his by the player until it is dead.

8.2 Critical Review

During the implementation of the prototype some problems emerged that were not expected.

- Mecanim is a relatively new system that was just released in 2013. Therefore, there are not many literature or tutorials for this topic which exacerbate the learnabilty of the game engine. From update to update, the system gets more stable. But it is not easy to teach this complex system to oneself.
- Mainly physics is a huge topic in game development and cater for several bugs referring collision detection. But implementing game physics would be an own huge subject. It is almost thinkable that the physics scripts that are provided by Unity work well but unfortunately, this is not the case.
- Project management is not easy to estimate for beginners. Occurring bugs were not included in milestones. That is because all team members were inexperienced designers and developers. In general, the effort was underestimated, but skill comes with practise.

8.3 Prospect

The prototype can be further developed. Primarily all bugs should be eliminated before adding new functionalities. Programmers should be concerned first with physics in order to understand and solve the problems. It should be good to implement physics on your own in order to reconstruct this subject.

Programming features like enemy AI (artificial intelligence) can be implemented afterwards. Also 3D enhancements can be improved meanwhile. The decision also depends of the game development team of *Scout COD*.

Summing up, this project was a haunting experience. With the cooperation of an enthusiast team, a satisfying prototype has been developed.

Part V

Appendix

Appendix A

Buglist

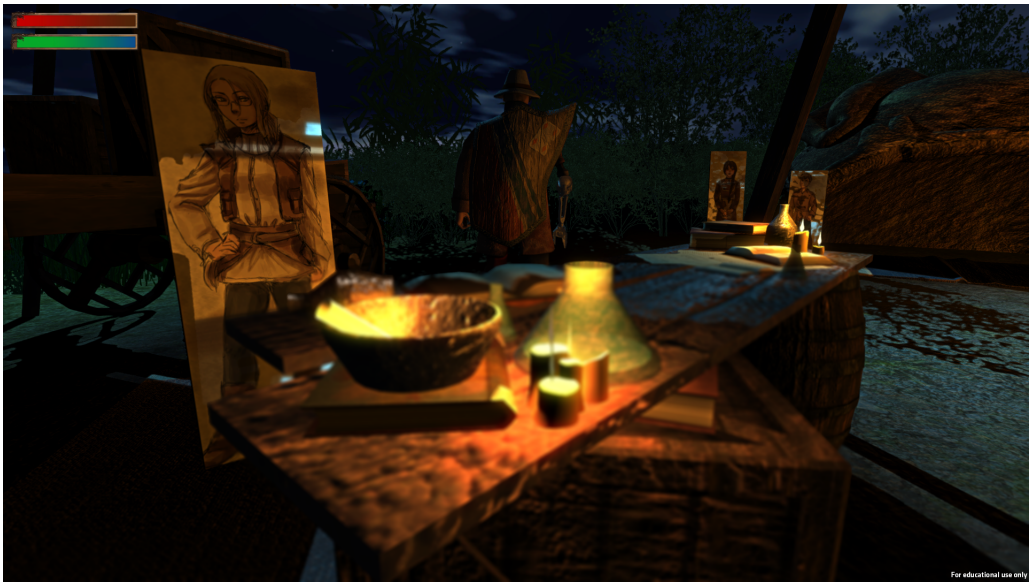
The prototype of *Scout COD* has the following bugs:

- If the player collides with an enemy, the Scout character sometimes start to fly and the enemies are bugging too.
- The Trigger zone is not always working.
- The attack chain does not work constantly.
- If the player is exhausted and wants to move the camera around the character, the camera is a bit jerking.
- If the player character is exhausted, it moves forwards for one single frame and then it turns in the right direction.
- 'Capslock', and thereby walking, is not working on MAC OS X.
- Sneaking with 'Ctrl' does not work in Unity but in the exported project. May be some short cuts inside of Unity disallows combining keys with 'Ctrl', for example*SneakBackwards* is not possible because 'Ctrl' + 'S' is a combination for saving.
- The character sometimes moves in the 3D world without any motion.

Appendix B

Screenshots

The following images are part of the prototype *Scout COD*.




For educational use only


For educational use only

List of Figures

Listings

List of Tables

Bibliography

Allen Sherrod: *Ultimate 3D Game Engine Design & Architecture*. Mason and Ohio: Cengage Learning, 2009

Balzert, Heide: *UML 2 kompakt: Mit Checklisten*. 2nd edition. München and Heidelberg: Elsevier, Spektrum, Akad. Verl., 2005, kompakt-Reihe

Bass, Len/Clements, Paul/Kazman, Rick: *Software architecture in practice*. 3rd edition. Upper Saddle River and NJ: Addison-Wesley, 2013, SEI series in software engineering

Buschmann, Frank et al.: *Pattern-orientierte Software-Architektur: Ein Pattern-System*. 1st edition. Bonn (u.a.): Addison-Wesley, 2000, Professionelle Softwareentwicklung

Crytek: *Crydev.net*. Frankfurt, 2014 ⟨URL: `http://www.crydev.net/`⟩ – visited on 24.03.2014

DBolical Pty Ltd: *Mod DB - Game Engines*. 2014 ⟨URL: `http://www.moddb.com/engines`⟩ – visited on 24.03.2014

Eilebrecht, Karl/Starke, Gernot: *Patterns kompakt: Entwurfsmuster für effektive Software-Entwicklung*. 2nd edition. München and Heidelberg: Elsevier, Spektrum, Akad. Verl, 2007, kompakt-Reihe

Epic Games: *Unreal Engine*. 2014 ⟨URL: `https://www.unrealengine.com/`⟩ – visited on 24.03.2014

GameFromScratch: *3D Game Engine Round-up*. 2011 ⟨URL: `http://www.gamefromscratch.com/page/3D-Game-Engine-Round-up.aspx`⟩ – visited on 24.03.2014

Gamma, Erich et al.: *Design patterns: Elements of reusable object-oriented software*. Reading and Mass: Addison-Wesley, 1995, Addison-Wesley professional computing series

Goldstone, Will: *Unity 3. x Game Development Essentials*. 2nd edition. Birmingham: Packt Publishing, Limited, 2011

Gregory, Jason: *Game engine architecture*. Wellesley and Mass: A K Peters, 2009

HOLZBAUER, Florian: *Grafik deluxe: Die besten Game-Engines der Welt*. 2010 ⟨URL: `http://www.chip.de/artikel/Grafik-deluxe-Die-besten-Game-Engines-der-Welt\_42976231.html`⟩ – visited on 24.03.2014

HUIZINGA, Johan: *Homo Ludens*. Reinbeck: Reclam, 2001

MENARD, Michelle: *Game development with Unity*. Boston and MA: Course Technology PTR, 2011

REHFELD, Gunther: *Game Design und Produktion: (Grundlagen, Anwendungen, Beispiele)*. München: Hanser, 2014, Hanser eLibrary

SALEN, Katie/ZIMMERMAN, Eric: *Rules of play: Game design fundamentals*. Cambridge and Mass: MIT Press, 2003

SCHELL, Jesse: *Die Kunst des Game Designs: (bessere Games konzipieren und entwickeln)*. 1st edition. Heidelberg (u.a.): mitp, 2012

SIX, Hans-Werner/LORENZ, Alexander/PILGRIM, Jens von: *Kurs 1794 - Software Engineering II, Methodische Entwicklung von Webapplikationen*. Ph. D thesis, FernUniversität in Hagen, Hagen, 2007

THORN, Alan: *Unity 4 fundamentals: Making games with Unity*. First edition edition. Burlington and MA: Focal Press, 2013

UNITY TECHNOLOGIES: *Unity 3D*. San Francisco, 2014 ⟨URL: `https://unity3d.com/`⟩ – visited on 05.03.2014

VALVE: *Valve Developer Community*. Bellevue, 2014 ⟨URL: `https://developer.valvesoftware.com/wiki/Source`⟩ – visited on 24.03.2014

WIKIPEDIA: *Liste von Spiel-Engines*. 2014 ⟨URL: `http://de.wikipedia.org/wiki/Liste\_von\_Spiel-Engines`⟩ – visited on 24.03.2014